Investigating Rust's usability and memory safety features by reimplementing known C/C++ bugs

Bruno Martins (bm2787)

Abstract

Systems software has long been dominated by the use of the low level systems programming languages C and C++: the Linux kernel, game engines and every major web browser are some of the most prominent examples of such systems. However, while C and C++ are the go-to languages to extract the most performance out of the hardware, they lack important safety features to prevent potentially catastrophic errors such as null pointer dereferences and buffer overruns. While such bugs in C/C++ code can potentially be detected by modern static analysis tools, Rust, a new programming language by Mozilla, claims to have eliminated entire classes of bugs at the language level by leveraging its unique type system. This paper looks into these claims, first by performing a review on a state-of-the-art static analyzer, Infer, then by evaluating the use of Rust in a real world stable production system, EPICS. Known race conditions, null dereferences, buffer overrun and other bugs in the EPICS framework were reimplemented in Rust to evaluate both Rust's ability to detect them and how easy such reimplementations would be. The reader is expected to learn about the Rust programming language, its strengths in bug prevention and the feasibility and potential pitfalls of migrating an existing C or C++ program into Rust.

Introduction

Mozilla has recently released a new systems programming language, Rust, that promises to make entire classes of bugs detectable and preventable at compile time [1]. Rust has a heavy focus on memory safety and uses novel concepts, like resource lifetimes and the borrow checker, to achieve that goal. However, these features 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. This study investigates Rust and attempts to provide answers to two important questions. First, does Rust prevent actual bugs? Second, is it straightforward to translate C and C++ programs into Rust?

In order to answer these questions, an established industrial and scientific controls system framework, EPICS, was chosen as a source for real C and C++ bugs. Issues in the EPICS framework issue tracker were triaged into ten distinct classes: six of which that were deemed of interest for this study (race condition, buffer overflow, use after free, type cast, null pointer dereference and return from stack) and four that were deemed out of scope (logic error, build system, feature request and bugs in third-party libraries). Then, one representative bug from each of the six issue classes was chosen to be reimplemented, first in a simpler form in its original language, then in Rust. Each version of the bug

was compiled and, when a binary was produced, executed, in order to observe both the compiler output and the executable behavior, so comparisons between compilers and languages could be drawn.

This paper is divided into several sections. This section, Introduction, presented the problem statement and the research method. The Background section has an overview of the chosen source of real bugs, the EPICS framework, and of the Rust programming language. It also has a review of a state-of-the-art static analyzer, Infer, and its theoretical underpinnings. The Methodology section describes in detail the issue classification process and the reimplementation and compilation processes. Further, the Results section gathers the results obtained in this study. The Conclusion section summarizes our findings and the answers to the proposed questions. Finally, the Future work section proposes further research that can be done on this topic.

Background

EPICS - Experimental Physics and Industrial Control System

The EPICS framework [2] is a set of open source software libraries and tools, written in C, C++ and Perl, designed to fulfill the role of a SCADA (Supervisory Control And Data Acquisition) system for big science facilities and industrial plants. EPICS has a distributed nature: its main component is the IOC (Input/Output Controller), which is responsible for communicating with physical devices and making their accepted commands and captured data available to the network via PVs (Process Variables). The PVs in an IOC are considered part of a Distributed Run Time Database. Other EPICS components, like graphical user interfaces, data archivers and user scripts, interact with the IOCs (and therefore the devices they control) by reading from and writing to the available PVs on the network, via one of two existing protocols: Channel Access [3] or PV Access [4]. Channel Access, the older of the two, is capable of transporting scalar and array values only, along with their metadata. The newer PV Access protocol, on the other hand, can transport arbitrarily structured data. A simplified diagram of EPICS' architecture is shown in Figure 1.

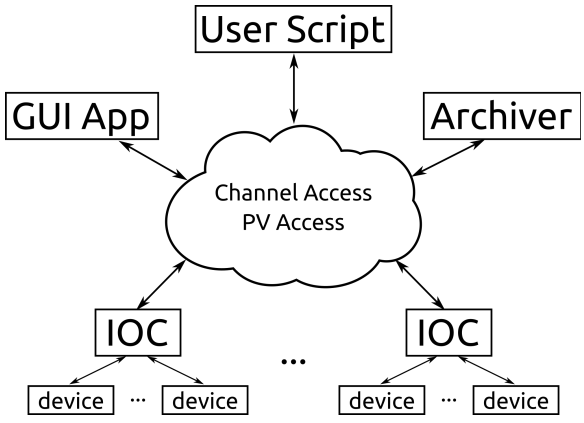

Figure 1: Simplified EPICS architecture

EPICS is used in different kinds of scientific facilities across the world: particle accelerators, observatories, telescope arrays, research fusion reactors and many others [5]. Given its widespread use and the critical role it performs, it is of paramount importance that its central component, the IOC, doesn't crash.

Rust

Rust is a new systems programming language, developed by Mozilla, focused on safety, speed and concurrency [1]. The first stable release of the language, dubbed version 1.0, was released on May 2015. Rust development was driven chiefly by the concurrent development of the Servo web browser engine, also by Mozilla, to eventually replace Gecko, their current rendering engine. Rust's advanced type system helped Servo developers to better address common security issues while still producing code with good performance and similar memory usage [6]. Like C++, Rust also promises to have zero-cost abstractions, or as it is commonly said, "you only pay for what you use", in order to produce fast compiled code. A simple introductory Rust program is shown in Listing 1.

```
fn main() {
    println!("Hello, world!");
}
```

Listing 1: Hello world in Rust

Rust's strength lies in its Hindley-Milner type system [7] [8] and its ownership system, which can be divided into three concepts: ownership, borrowing and lifetimes. These concepts play a fundamental role in ensuring memory safety.

The **ownership** concept simply ensures that values in a 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 2, 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 ownership constraints, so the compiler prevents it from happening.

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

```
$ cargo build
   Compiling example1 v0.1.0 (/home/bmartins/comsw6156/paper/examples/ownership)
<-- warning snipped -->

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
```

Listing 2: Ownership example

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 is instead copied to the destination. All primitive types implement the `Copy` trait.

If non-`Copy`able 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**. An example of this mechanism is shown in Listing 3.

```
fn print_vec(v: &Vec<i32>) {
    println!("{:?}", v);
}

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

```
$ cargo run
   Compiling borrow v0.1.0 (/home/bmartins/comsw6156/paper/examples/borrow)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target/debug/borrow`
[1, 2, 3]
[1, 2, 3]
```

Listing 3: 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.

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 they can be made explicit, as shown in a contrived example in Listing 4, where a function `sum` takes two parameters by reference: `x`, with lifetime `a`, and `y`, with lifetime `b`. Rust ensures that both lifetimes are long enough to be valid inside the function.

```
fn sum<'a, 'b>(x: &'a i32, y: &'b i32) -> i32 {
    return x + y;
}
```

Listing 4: Explicit lifetimes example

Lifetimes are a way of giving names to scopes. So, essentially, the compiler ensure that the references needed in different functions don't go out of scope, preventing use-after-free and similar bug classes.

Rust has many other nice features not mentioned in this section, usually only available in higher level or functional languages: first-class functions, closures, algebraic types, etc. Since Rust is a fairly new language, there is a considerable amount of research on it; a few works will be mentioned. There is an effort to verify Rust's type checker using fuzzing [9]; a work on applying separation logic (same theoretical basis as used for Infer, described in the next section) to verify the compiler [10]; exhaustive test case generation for `unsafe` blocks in a Rust program [11]; regular Rust program verification through symbolic execution [12] and, more recently, a full formalization of its type system called Oxide [13].

Infer, a state of the art tool for static analysis

Static analyzers are invaluable tools to prevent bugs from occurring in production code. While compiled languages typically have their compilers do some static analysis, it is usually not sufficient. For example, as will be demonstrated later in this paper, gcc and g++ are not capable of detecting many bug classes that can cause a program to return incorrect results, to crash, or even to leave it open to malicious attacks. Therefore, it is for good reason that research on static analysis continues to evolve.

Infer, a recent static analysis tool built in academia and brought to Facebook by its creators [14], will be briefly reviewed in this section. Infer builds on Hoare logic and separation logic theories in order to parse the program under analysis into a simpler, symbolic representation, which can then be executed by Infer's symbolic interpreter. The symbolic interpreter looks for memory violations according to its model and presents the ones it finds to the user.

The fundamental operation that Infer performs to build the symbolic representation of a program is what their authors called *bi-abduction*. Essentially, for a piece of code in isolation, called `C`, Infer infers the separation logic [15] formulae in the pre-condition `P` and in the post-condition `Q` in the Hoare triple [16]:

$$\{P\}C\{Q\}$$

In particular, the pre- and post-conditions describe the heap of the program before and after `C` executes. By solving this problem, Infer finds the *frame* `F` for `C` (its footprint). Since the analysis is performed locally, as opposed to performing whole-program analysis, Infer can be run incrementally, greatly reducing overhead.

Furthermore, Infer is extensible: a recently published extension, FootPatch [17], takes advantage of Infer's symbolic representation of programs to automatically find repairs to memory violations `{P} C {Q}` by *searching* everywhere in the program under analysis for a code segment `C'` that has a precondition `P'` similar to `P` but a non-violating post-condition `Q'`. If FootPatch finds such `C'`, it suggests replacing `C` with `C'`, while maintaining local characteristics, like variable names, intact.

Another advantage of Infer is that, since it looks for universal memory violations in the symbolic representation of a program, it can be run on different programming languages: C, C++, Objective-C and Java are supported. Therefore, any improvements to Infer, like FootPatch, are automatically valid for all of the supported programming languages.

Nonetheless, as remarkable as Infer is, there are certain limitations. Despite being classified as a *static* analysis tool (and it is so by the point of view of the original source code), Infer has to execute the *symbolic* representation of the program, which can contain infinite loops and other conditions that can cause the analysis to be skipped for certain regions of the code. Furthermore, Infer can't detect concurrency issues and dynamic dispatches in Android and iPhone apps. Finally, Infer is a third-party tool that has to be specifically configured and executed as part of the build process of a project, which can potentially hinder adoption.

Rust claims to be able to detect all bug classes that Infer can detect, and more, and it does so during the ordinary compilation process, automatically.

Methodology

In order to evaluate Rust's safety features, an existing mature open source project, EPICS, was chosen to be the subject of this study. First, a custom tool was developed to help analyze and classify issues in the EPICS' issue tracker, hosted in Launchpad. The issues, selected from the pool of issues with existing fixes, were classified into ten broad classes, which will be described in the next section. Then, 6 individual bugs from different classes were chosen to be reimplemented in Rust, in order to test Rust's ability to detect the existing known bugs in the code.

Issue classification

EPICS, being almost three decades old, has a long history in its issue tracker. Currently, EPICS' core components use Launchpad as their issue tracker, having migrated from Mantis circa 2002. Other, newer components live on GitHub, and use GitHub's issue tracker. In this study, only Launchpad's issues were considered.

Since the goal of this paper is to evaluate Rust's ability to catch bugs, the issues were filtered by the tags "Fix Committed" and "Fix Released", so each issue can be attributed to a class with more certainty. As of the writing of this paper, there were 413 issues that had either the "Fix Committed" or the "Fix Released" tag. In order to help sift through and classify these issues in a reasonable amount of time, a custom classifier tool was developed to query Launchpad's servers via its API, present the information in a web page in a convenient form and store the classification information in a relational database, so it could be queried later. A screenshot of this tool is shown in Figure 2.

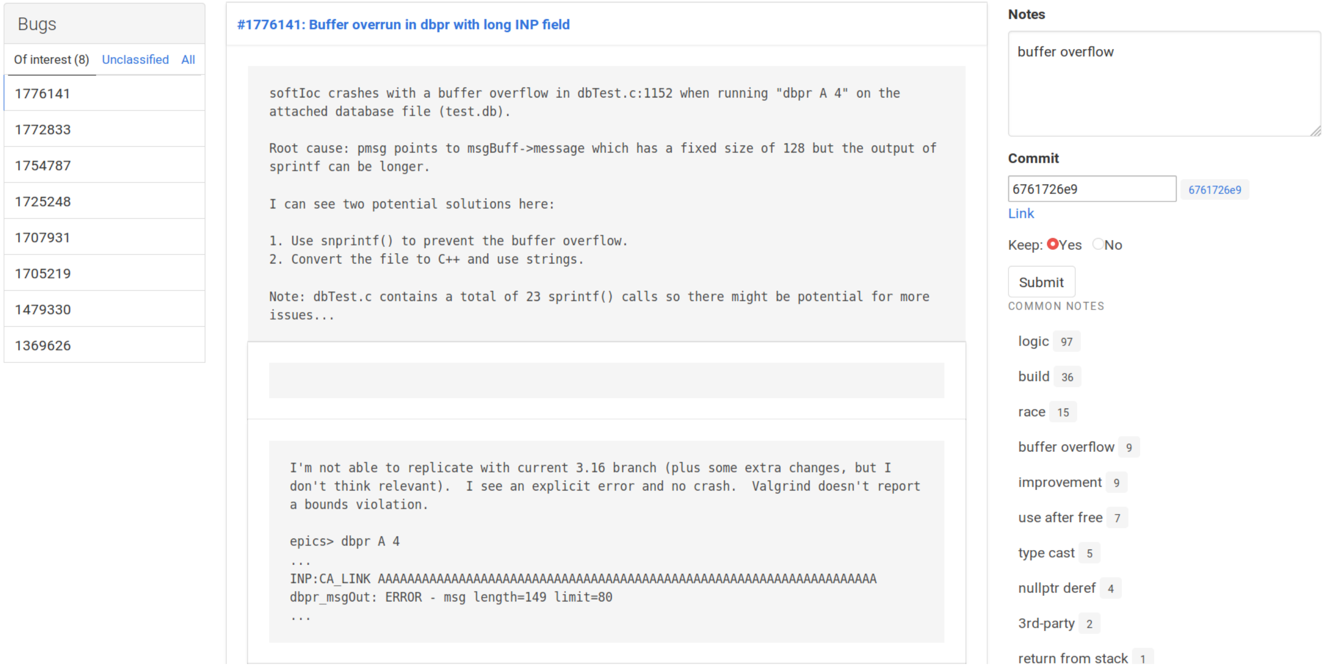

Figure 2: Custom issue classifier

The issue classifier presents a list of issue ID numbers on the left column, ordered from most to least recent, according to one of three categories: *All*, containing all available issues with "Fix

Committed" or "Fix Release" tags; *Unclassified*, with issues that were not given any classification by the author of this paper; and *Of Interest*, which are a subset of the classified issues that were to be potentially reimplemented in Rust for this study.

The center column contains the issue's ID and title, with an hyperlink to the issue's page on Launchpad, and the issue's description and comments below it.

On the right column there are a few fields of interest. The first one, named "Notes", is a string corresponding to the issue classification given by the author of this paper. The second one, "Commit", contains the commit hash in which the issue was fixed. The tool automatically looks at the version control history of the EPICS repository and searches for the issue ID in the commit messages. If such a commit is found, the tool presents the commit hash as a suggestion, to the right of the input text box, that can be clicked to automatically fill the input box. If a commit hash is present in the relational database, the tool also presents a "Link" hyperlink to that commit's page on the GitHub mirror of the EPICS project. The last field, named "Keep", serves as an indication of whether that particular issue will be put in the Of Interest group. By clicking the "Submit" button, the user causes the information that was entered to be committed to the database. Below the "Submit" button there is a list of classifications by the author from other issues, presented in decreasing order of use. If the user clicks on a particular item from this list, the item's text is copied to the "Notes" field.

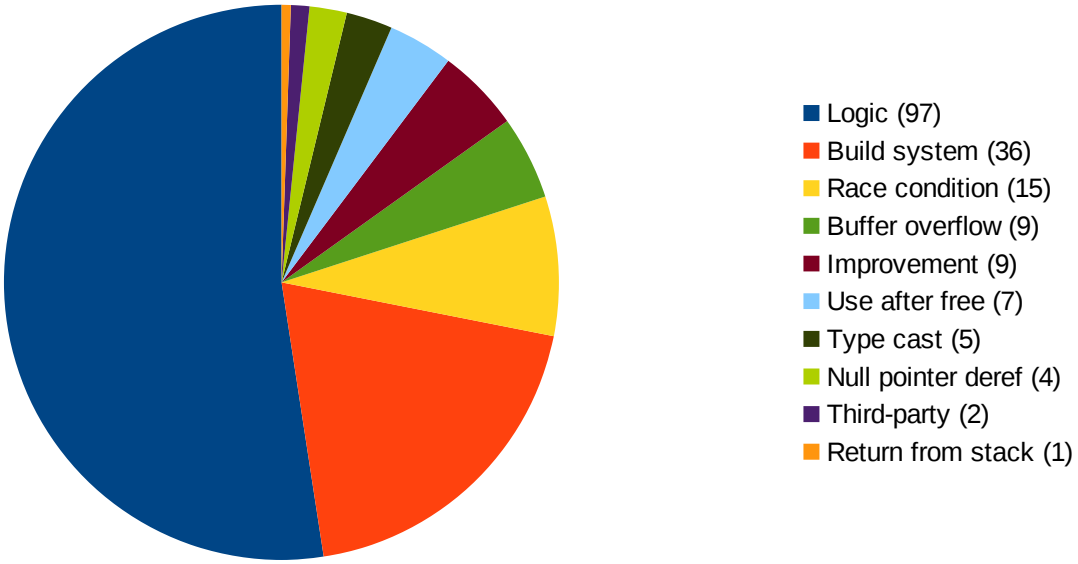

Figure 3: Issue classification chart

This custom tool enabled the author to classify the most recent 185 issues, spanning almost 10 years, in a reasonable time frame. Each issue was put in one of ten categories, as shown in Figure 3, at the author's discretion:

1) **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. There were 97 (52.4 %) such issues.

2) **Build system**: the EPICS framework predates build systems such as CMake. Therefore, EPICS authors had to roll out their own build system, built on top of Makefiles, that can have different

target platforms (Windows, Linux, RTEMS, vxWorks, etc). Issues in this class refer to bugs in the EPICS build system itself, and are not of interest for this study. There were 36 (19.4 %) such issues.

3) **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. There were 15 (8.1 %) such issues.

4) **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. There were 9 (4.8 %) such issues.

5) **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. There were 9 (4.8 %) such issues.

6) **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". There were 7 (3.7 %) such issues.

7) **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. There were 5 (2.7 %) such issues.

8) **Null pointer dereference**: issues of this nature refer to code that tries to dereference a pointer that was set to NULL. Rust can statically detect some cases of null dereference. There were 4 (2.1 %) such issues.

9) **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. There were 2 (1.0 %) such issues.

10) **Return from stack**: the single 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. There was 1 (0.5 %) such issue.

Bug reimplementation

As noted throughout the previous section, 6 classes of issues were of interest to this study: race condition, buffer overflow, use after free, type cast, null pointer dereference and return from stack. One representative bug from each class was chosen to be reimplemented 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. 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 writing.

Buffer overflow bug

C

Source Code

```
#include <stdio.h>

typedef struct msgBuff { char message[128]; }
TAB_BUFFER;

int main(void) {
    TAB_BUFFER buf;
    char *type = "dummy";
    char *pdbentry = "dummy";

    // Write >128 chars
    char *pfield_name =
        "12345678901234567890123456789O"
        "12345678901234567890123456789O"
        "12345678901234567890123456789O"
        "12345678901234567890123456789O"
        "12345678901234567890123456789O";

    sprintf(buf.message, "%-4sL %s %s",
        pfield_name, type, pdbentry);

    return 0;
}
```

Compilation and Execution Results

```
$ gcc -Wall -Wextra main.c -o main

$ ./main
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

Rust

Source Code

```
struct MsgBuff { message: [u8; 128] }

fn main() {
    let mut m = MsgBuff {
        message: [0; 128]
    };

    let type_ = "dummy";
    let pdbentry = "dummy";
    let pfield_name =
        "12345678901234567890123456789O\
         12345678901234567890123456789O\
         12345678901234567890123456789O\
         12345678901234567890123456789O\
         12345678901234567890123456789O";

    let formatted = format!("{:<4}L {} {}",
        pfield_name, type_, pdbentry);
    m.message.copy_from_slice(
        formatted.as_bytes());
}
```

Compilation and Execution Results

```
$ cargo build
      Compiling bug1776141 v0.1.0 (/home/bmartins/comsw6156/paper/
rust/bug1776141)
    Finished dev [unoptimized + debuginfo] target(s) in 0.21s

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/bug1776141`
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `128`,
 right: `163`: destination and source slices have different
lengths', src/libcore/slice/mod.rs:1965:9
note: Run with `RUST_BACKTRACE=1` environment variable to display
a backtrace.
```

Listing 5: C and Rust code, compilation trace and execution trace for simplified issue #1776141

A buffer overflow bug consists of writing to a buffer past its end. This kind of bug can lead to security vulnerabilities in which an attacker can potentially take control of the program. C and C++ do

not check bounds on array reads or writes since these operations get translated to pointer arithmetic, which are not bounds checked at all, as Kernighan and Ritchie put it:

> Rather more surprising, at least at first sight, is the fact that a reference to `a[i]` can also be written as `*(a+i)`. In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are completely equivalent. [18]

The EPICS issue #1776141 was chosen as an example of this class of bugs. In this bug, a call to `sprintf` was being made without checking for the buffer size beforehand, as seen in Listing 5. By writing past `message` bounds, the C code caused the program to be immediately aborted and core dumped. Modern gcc versions introduce a check for this particular condition in the binary (shown as "stack smashing detected" in the program's execution trace), but such check is introduced entirely at the compiler level, not at the language level. Also, the process of aborting the C program occurs brusquely and the programmer has to figure out the issue at hand by analyzing the core dump using different tools, like gdb, which can be laborious.

Rust, in this case, didn't catch the bug at compile time because the compiler cannot statically prove that the buffer being written to will be large enough. However, the function `copy_from_slice` does perform bounds check and can tell exactly what went wrong before panicking (Rust's term for aborting). This is arguably safer since the check is being performed by a language construct and does not depend on the compiler implementation.

Use after free bug

	Source Code	Compilation and Execution Results

C++	`#include <stdio.h>`
	
	`struct Dummy {`
	`  int d;`
	`  Dummy(int d):d(d) {}`
	`};`
	
	`int main(void) {`
	`  Dummy *dummy = new Dummy(42);`
	`  delete dummy;`
	`  printf("%d\n", dummy->d);`
	`  return 0;`
	`}`
		`$ g++ -Wall -Wextra main.cpp -o main`
		
		`$ ./main`
		`0`

Rust	`#[derive(Debug)]` `struct Dummy(i32);` `fn main() {` `  let a = Dummy(42);` `  drop(a);` `  println!("{:?}", a);` `}`	**`$ cargo build`** `  Compiling bug861214 v0.1.0 (/home/bmartins/comsw6156/paper/rust/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`  ***`<-- remainder of message snipped -->`***

Listing 6: C++ and Rust code, compilation trace and execution trace for simplified issue #861214

One example of a 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 6.

These programs illustrates 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 all 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 *transferred* 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 and then does nothing.

Type cast bug

The EPICS issue #1707931 contains a type cast bug. The real bug was triggered in only one of many platforms that particular snippet was being compiled to. It didn't cause a crash in that particular instance, but led unnecessary reprocessing of network messages. Type cast bugs can be surprising to the programmer, given the complicated rules for implicit type conversion in C and C++. In this particular example, gcc is shown to emit warnings for the meaningless comparisons, but compiles the program anyway, as shown in Listing 7.

Rust, on the other hand, is much more strict about type conversions. Here, it disallows the compilation to continue when it detects that a signed value is being assigned to an unsigned variable. However, it does not warn about the meaningless first comparison, which can presumably be optimized away by the compiler, and might indicate a logic bug in the program.

	Source Code	Compilation and Execution Results

C	`#include <stdio.h>`
	
	`int main(void) {`
	`  unsigned int u = 42;`
	`  if (u < 0)`
	`  printf("Should never happen");`
	
	`  int i = -42;`
	`  unsigned int u2 = i;`
	`  if (u2 < 0)`
	`  printf("Should never happen");`
	
	`  return 0;`
	`}`
		$ gcc -Wall -Wextra main.cpp -o main
		main.c: In function ‘main’:
		main.c:5:11: warning: comparison of unsigned expression < 0 is always false [-Wtype-limits]
		  if (u < 0)
		  ^
		main.c:10:12: warning: comparison of unsigned expression < 0 is always false [-Wtype-limits]
		  if (u2 < 0)
		  ^
		
		$ ./main
		$

Rust	`fn main() {`
	`  let u = 42u32;`
	`  if u < 0 {`
	`  println!("Should never happen");`
	`  }`
	
	`  let i = -42i32;`
	`  let u:u32 = i;`
	`  if u < 0 {`
	`  println!("Should never happen");`
	`  }`
	`}`
		$ cargo build
		  Compiling bug1707931 v0.1.0 (/home/bmartins/comsw6156/paper/rust/bug1707931)
		error[E0308]: mismatched types
		 --> src/main.rs:8:17
		  |
		8 | let u:u32 = i;
		  | ^ expected u32, found i32
		
		<-- remainder of message snipped -->

Listing 7: C and Rust code, compilation trace and execution trace for simplified issue #1707931

Null pointer dereference bug

Null references were once called a "billion dollar mistake" by Sir Anthony Hoare [19], given how dangerous and costly they can be. The EPICS issue #1369626 has such a bug: EPICS allows 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 dereference, as seen in Listing 8.

	Source Code	Compilation and Execution Results

C	`typedef void caEventCallbackFunc(int arg);`
	
	`int main(void) {`
	`  caEventCallbackFunc *cb = NULL;`
	`  cb(0);`
	`  return 0;`
	`}`
		`$ gcc -Wall -Wextra main.c -o main`
		
		`$ ./main`
		`Segmentation fault (core dumped)`

Rust	`type CaEventCallbackFunc =`
	`  *const fn (i32) -> ();`
	
	`fn main() {`
	`  let cb : CaEventCallbackFunc =`
	`  std::ptr::null();`
	`  (*cb)(0);`
	`}`
		`$ cargo build`
		`  Compiling bug1369626 v0.1.0 (/home/bmartins/comsw6156/paper/rust/bug1369626)`
		`error[E0133]: dereference of raw pointer is unsafe and requires unsafe function or block`
		`--> src/main.rs:5:5`
		`  |`
		`5 |  (*cb)(0);`
		`  |  ^^^^^ dereference of raw pointer`
		`  |`
		`  = note: raw pointers may be NULL, dangling or unaligned; they can violate aliasing rules and cause data races: all of these are undefined behavior`
		
		`<-- remainder of message snipped -->`

Listing 8: C and Rust code, compilation trace and execution trace for simplified issue #1369626

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 dereferenced, but the error message tells us that the dereference 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 checks for that particular snippet. From Rust's perspective, gcc can be thought of as running in unsafe mode all the time.

Return from stack bug

Returning data by reference from a function's stack is a bug because a function's stack is ephemeral: as soon as the function ends its execution, the stack is deallocated.

The EPICS issue #1705219 contains such a bug. The real bug was not as obvious as the simplification in Listing 9; a function was being called with a reference to stack data, but it wasn't obvious from the context that the callee was launching a separate thread to complete its work. The separate thread then outlived the caller stack, causing a segmentation fault.

gcc is capable of detecting the bug in the example code, but does not halt compilation. Executing the generated binary fatally leads to a segmentation fault.

	Source Code	Compilation and Execution

C	`#include <stdio.h>`
	
	`char *get_str(void) {`
	`  char s[64] = "dummy";`
	`  return s;`
	`}`
	
	`int main(void) {`
	`  char *s = get_str();`
	`  printf("%s\n", s);`
	`  return 0;`
	`}`
		`$ gcc -Wall -Wextra main.c -o main`
		`main.c: In function ‘get_str’:`
		`main.c:5:12: warning: function returns address of local variable [-Wreturn-local-addr]`
		`  return s;`
		`  ^`
		
		`$ ./main`
		`Segmentation fault (core dumped)`

Rust	`fn get_str() -> &str {` `  let s:&str = "dummy";` `  return s;` `}` `fn main() {` `  let s = get_str();` `  println!("{}", s);` `}`	**`$ cargo build`** `  Compiling bug1705219 v0.1.0 (/home/bmartins/comsw6156/paper/rust/bug1705219)` `error[E0106]: missing lifetime specifier` `--> src/main.rs:1:17` `  |` `1 | fn get_str() -> &str {` `  |  ^ help: consider giving it a 'static lifetime: `&'static` ` |` ` = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from`  ***`<-- remaining messages snipped -->`***

Listing 9: C and Rust code, compilation trace and execution trace for simplified issue #1705219

Rust, once again, can catch this bug at compile time. It does this by using its concept of lifetimes: it knows that the data backing up the string will go away when the function finishes executing, and that future references to that string will be invalid. Hence, it halts the compilation and emits a helpful message about what happened and one possible solution to the issue.

Race condition bug

Race conditions are susceptible to happen in concurrent code. If one or more threads try to access a resource without proper synchronization, subtle logic issues can be introduced. If a thread releases a resource being used by another thread, a crash can occur. While the analysis done on the EPICS issue tracker turned up 15 race condition bugs, they were often too subtle or too complicated to be presented in a succint form here. Instead, Listing 10 presents a synthetic example of ten threads simultaneously incrementing a single counter by one.

At a first glance, the C++ code looks correct. g++ does not have any objections and the execution trace shows a successful, clean execution. However, the race condition bug is subtle: the variable `counter` is not protected by any synchronization object. Since the addition assignment operator (+=) performs a read-modify-write operation, the C++ code as written is susceptible to producing an incorrect result if two or more of the ten threads are context-switched out after reading the contents of `counter` but before being able to increment it.

	Source Code	Compilation and Execution Results

C++	`#include <thread>`
	`#include <vector>`
	
	`void incr(int* counter) {`
	`  *counter += 1;`
	`}`
	
	`int main(void) {`
	`  int counter = 0;`
	`  std::vector<std::thread> handles;`
	
	`  for (int i = 0; i < 10; ++i)`
	`  handles.push_back(std::thread(incr, &counter));`
	
	`  for (auto& handle : handles)`
	`  handle.join();`
	
	`  printf("Result: %d\n", counter);`
	`}`
		$ g++ -Wall -Wextra main.cpp -o main -lpthread
		
		$ ./main
		Result: 10

Rust	`use std::thread;` `fn main() {` `  let mut counter = 0;` `  let mut handles = vec![];` `  for _ in 0..10 {` `  let handle = thread::spawn( || {` `  counter += 1;` `  });` `  handles.push(handle);` `  }` `  for handle in handles {` `  handle.join().unwrap();` `  }` `  println!("Result: {}", counter);` `}`	**$ cargo build** `  Compiling bugrace v0.1.0 (/home/bmartins/comsw6156/paper/rust/bugrace)` `error[E0499]: cannot borrow `counter` as mutable more than once at a time` `  --> src/main.rs:8:37` `  |` `8	let handle = thread::spawn(|| {` ` | - ^^ mutable borrow starts here in previous iteration of loop` ` | _____|` ` | |` `9 | | counter += 1;` ` | | ------- borrows occur due to use of `counter` in closure` `10 | | });` `	|_____- argument requires that `counter` is borrowed for `'static`` ***<-- remainder of message snipped -->***

Listing 10: C++ and Rust code, compilation trace and execution trace for a race condition bug

Rust catches this bug through its so-called borrow checker. An immutable variable in Rust can be "borrowed" many times, since its value is effectively constant. Any number of threads can read an immutable value without side effects. A mutable variable, on the other hand, is only allowed to be "borrowed" once. In this particular case, Rust detects that there are several "borrows" (the ten threads) and does not allow the compilation to proceed, effectively preventing the bug from being compiled.

Results

For this paper, 185 issues on the EPICS project Launchpad issue tracker were classified, using a customized tool, out of the 413 issues marked as fixed available at the time of writing, into 10 distinct classes. Most of the classified issues, 144 (77.8%), belonged to 4 classes 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. 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. That was accomplished for 5 classes: buffer overflow, use after free, type cast, null dereference and return from stack. The last issue class, race condition, didn't have a bug that was straightforward to be reimplemented in a short program. Therefore, a synthetic representative multi-threaded program was used instead. The results from the observed behavior from the compilers and compiled programs involved are summarized in Table 1.

Bug class	Language	Bug caught at		Execution Crashed	Incorrect Result
		Compilation	Execution		
Buffer overflow (9)	C			**Yes**	
	Rust		**Yes**		
Use after free (7)	C++				**Yes**
	Rust	**Yes**			
Type cast (5)	C	**Warning**			
	Rust	**Yes**			
Null dereference (4)	C			**Yes**	
	Rust	**Yes**			
Return from stack (1)	C	**Warning**		**Yes**	
	Rust	**Yes**			
Race condition (15)	C++				
	Rust	**Yes**			

Table 1: Results summary

Bug detection performance

The first goal of this study was to evaluate the bug detection capabilities of Rust in contrast with two other major systems languages, C and C++, in the context of an established framework written in those languages. In that regard, Rust was able to remarkably catch *all* 6 tested bugs, using the default out of the box compiler options: 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 the 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 written in Rust, it would be not 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 (buffer

overflow) class 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 their warnings can vary greatly by platform and compiler version, potentially making the build system brittle. It is also interesting to note that 2 bugs that didn't even get a warning emitted by gcc resulted in an outright crash at runtime.

Rust's usability

The second goal of this study was to evaluate the usability of the Rust language with respect to its learning curve and to the ease of porting code from C and C++. Surprisingly (to the author), Rust was very straightforward to install and simple to start coding with. The Rust book [20] 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.

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 like null pointer dereference, 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.

A review of a particular static analysis tool, Infer, was performed. Infer advances the state-of-the-art static analyzer by relying on separation logic to symbolically execute the program under analysis and detect faulty behavior. While this is an important advance to the field, there are unavoidable limitations, especially in face of the undecidability of the halting problem [21], which prevents Infer from analyzing code sections that might take too long to execute or never finish.

An alternative approach to prevent certain classes of bugs is to design a language in a way that such bugs become impossible to be written. This is the approach taken by the Rust programming language. A review of Rust's unique ownership concepts was presented.

In order to investigate Rust's safety features, 185 issues from a established controls system framework, EPICS, were classified into 10 different categories. Then, 6 of these categories were selected as candidates of bug classes that could be prevented by Rust. One representative issue from

each of the selected categories was chosen to be reimplemented in a simplified form in C or C++ and Rust. Finally, the compiler outputs for both the original code in C or C++ and the translated Rust code were analyzed, as well as the resulting compiled program's outputs, when compilation succeeded.

From the safety point of view, Rust was able to prevent 5 out of the 6 issues from being compiled and the sixth one from causing a segmentation fault, clearly demonstrating that Rust's safety features work well. From the usability point of view, all translations from C or C++ into Rust were straightforward and didn't require much effort. However, every translated piece of code was short and simple, which certainly cannot lead to the conclusion that Rust is easy to write in general.

I learned from this study that Rust is a promising new language that is easier to get started with than I anticipated. Its official documentation is good and thorough, and made me want to start to using it for personal and professional projects. Its safety features work incredibly well, its build system (cargo) is powerful and simple to use and its higher level constructs are enticing, especially to me as a C and C++ programmer.

Future work

Given Rust's clear advantages over C and C++ regarding memory safety, a broader study could be done regarding Rust's usability: automatic or software-guided translation from C and C++ into Rust, especially for critical software, could be investigated. The author plans to choose a module from the EPICS framework to be rewritten in Rust so both its broader usability and its interoperability capabilities with C and C++ can be investigated.

References

[1] N. D. Matsakis and F. S. Klock II, “The Rust Language,” in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, New York, NY, USA, 2014, pp. 103–104.

[2] L. Dalesio, A. Kozubal, and M. Kraimer, “EPICS architecture,” Los Alamos National Lab., NM (United States), LA-UR-91-3543; CONF-911116-9, Jan. 1991.

[3] J. Hill and R. Lange, “EPICS Channel Access 4.13.1 Reference Manual,” 2009. [Online]. Available: https://epics.anl.gov/base/R7-0/2-docs/CAref.html. [Accessed: 24-Feb-2019].

[4] M. Sekoranja *et al.*, “pvAccess Protocol Specification,” 16-Oct-2015. [Online]. Available: http://epics-pvdata.sourceforge.net/pvAccess_Protocol_Specification.html. [Accessed: 24-Feb-2019].

[5] “Projects - EPICS Controls,” *EPICS Controls*. [Online]. Available: https://epics-controls.org/epics-users/projects/. [Accessed: 24-Feb-2019].

[6] B. Anderson *et al.*, “Engineering the Servo Web Browser Engine Using Rust,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, New York, NY, USA, 2016, pp. 81–89.

[7] R. Hindley, “The Principal Type-Scheme of an Object in Combinatory Logic,” *Trans. Am. Math. Soc.*, vol. 146, pp. 29–60, 1969.

[8] R. Milner, “A theory of type polymorphism in programming,” *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, Dec. 1978.

[9] K. Dewey, J. Roesch, and B. Hardekopf, “Fuzzing the Rust Typechecker Using CLP (T),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 482–493.

[10] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “RustBelt: Securing the Foundations of the Rust Programming Language,” *Proc ACM Program Lang*, vol. 2, no. POPL, pp. 66:1–66:34, Dec. 2017.

[11] J. Toman, S. Pernsteiner, and E. Torlak, “Crust: A Bounded Verifier for Rust (N),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 75–80.

[12] M. Lindner, J. Aparicius, and P. Lindgren, “No Panic! Verification of Rust Programs by Symbolic Execution,” in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018, pp. 108–114.

[13] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, “Oxide: The Essence of Rust,” *ArXiv190300982 Cs*, Mar. 2019.

[14] C. Calcagno *et al.*, “Moving Fast with Software Verification,” p. 9.

[15] J. C. Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures,” in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, USA, 2002, pp. 55–74.

[16] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Commun ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[17] R. van Tonder and C. L. Goues, “Static Automated Program Repair for Heap Properties,” in *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA, 2018, pp. 151–162.

[18] B. W. Kernighan and D. M. Ritchie, “Pointers and Arrays,” in *The C Programming Language*, 1st ed., Prentice-Hall, 1978, p. 94.

[19] T. Hoare, “Null References: The Billion Dollar Mistake,” presented at the QCon London, 2009.
[20] J. Blandy, *The Rust Programming Language: Fast, Safe, and Beautiful*. O’Reilly Media, Inc., 2015.
[21] A. M. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem,” *Proc. Lond. Math. Soc.*, vol. s2-42, no. 1, pp. 230–265, 1937.