CERES
Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
Ediciones Clío
Año 1, núm. 2, julio-diciembre, 2026
ISSN-e: 3101-4895

Sección: **Artículo científico** | **2026**, julio-diciembre, año 1, núm. 2, 34-69

Adding a Rust-like memory ownership model to classic C language

DOI: https://doi.org/10.5281/zenodo.21157881

Costas Rodríguez, Sergio[1]
Correo: rastersoft@gmail.com
Orcid: https://orcid.org/0009-0006-1940-957X
Canonical. Reino Unido

Abstract

RUST programming language is becoming an inter- esting alternative to C thanks to its memory ownership model, which allows to write safer code without a runtime system or a garbage collector. This increases the stability and speed without penalty in code size or CPU usage. Unfortunately, as a relatively new language, it forces developers to learn and master it before being able to take full advantage of such capabilities. In this paper we present a C static analyzer that allows to implement some parts of that same memory ownership model in standard C code, allowing developers to create safer code (with less memory leaks and core dumps) without having to learn a new language or adding any kind of runtime code.

Keywords: C Language, Rust Language, static analyzer.

Incorporación de un modelo de propiedad de memoria similar al de Rust al lenguaje C clásico

Resumen

El lenguaje de programación RUST se está convirtiendo en una alternativa interesante a C gracias a su modelo de propiedad de memoria, que permite escribir código más seguro sin un sistema de tiempo de ejecución ni un recolector de basura. Esto aumenta la estabilidad y la velocidad sin aumentar el tamaño del código ni el uso de la CPU. Desafortunadamente, al ser un lenguaje relativamente nuevo, obliga a los desarrolladores a aprenderlo y dominarlo antes de poder aprovechar al máximo estas capacidades. En este artículo presentamos un analizador estático de C que permite implementar algunas partes de ese mismo

[1] Ing. en Telemática. Doctor en Ingeniería en Telecomunicaciones. Canonical. Reino Unido.

https://revistaceres.com/index.php/ceres/index
Recibido: 2026-02-05 **Aceptado**: 2026-04-10

modelo de propiedad de memoria en código C estándar, lo que permite a los desarrolladores crear código más seguro (con menos fugas de memoria y volcados de memoria) sin tener que aprender un nuevo lenguaje ni añadir ningún tipo de código de tiempo de ejecución.

Palabras clave: Lenguaje C, Lenguaje Rust, analizador estático.

Introduction

RUST[1] is a relatively new *system programming language* created originally by Mozilla Research, part of the Mozilla Foundation[2] and currently developed by the Rust foundation[3], that supports both functional and imperative-procedural paradigms. It aims to offer similar performance to C/C++, while being safer thanks to implement a strict memory model to manage the allocated memory blocks. This allows to ensure that there won't be neither *dangling pointers*, nor *memory leaks*.

Several projects born as RUST-only projects, like Servo[2][4] and Redox[3][5], but it is also gaining traction in other projects, like the Gnome Desktop[6][7].

Gnome Desktop started as an alternative to the KDE Desktop[8]. From the beginning, the developer team was aware of the inherent problems of the C language, and that is why they decided to offer bindings for as many languages as possible, allowing to create *first-class* applications for Gnome using Perl, Python, C++, C#, Java, Lua... To simplify this task even further, they developed *GObject Introspection*[4]. The last iteration (Gnome 3) goes as far as to use Javascript for the desktop itself. But no matter how many elements are developed in high-level

[2] Servo is a browser engine originally developed by Mozilla to replace the Gecko engine in Firefox. Although several components, like the CSS subsystem, have been backported to Gecko, currently it is no longer used for Firefox, and it is being developed independently

[3] Redox is an UNIX-like operating system being written in RUST

[4] GObject Introspection is a middleware between a GObject library and language bindings that allows to generate automatically the later. This elim- inates the need for manually writing the bindings for every new object or method in GObject-based libraries for each and every language, allowing to have up-to-date bindings with very little effort

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

languages, there will always be parts that must be written in C language. In the case of Gnome, these are, at least, the base libraries (GTK, Clutter, Gstreamer...) and the window manager (Mutter[5]).

The problem with GObject libraries relies in that they are as prone as any other C code to suffer memory leaks and dangling pointers. Some alternatives where proposed, like Vala[6], but none has really caught enough momentum as to be considered *the* solution.

Here is where RUST makes its appearance: being a language aiming to offer similar performance to C/C++ and being memory safe, it is ideal to replace C in libraries. In fact, it is being used to rewrite some libraries like *librsvg*[11] and to write new Gnome applications.

Another field where C is still very common is the microcontroller arena. Today, only the smallest microcontrollers are programmed in Assembly language, and it is common that the manufacturer even tries to enforce using C for programming their devices. But recently, RUST have been gaining momentum in this area, with microcontroller families like the STM32, the ESP32 and more.

Unfortunately, this still presents some problems:

- RUST is a new language, and its syntax is quite different from classic languages like C, so it requires learning and training before a programmer becomes competent in it.

- Porting a library to RUST implies to rewrite the code. Although it is possible to do it in a *function-by-function* manner, replacing only one C function with a new RUST one each time, it still implies throwing all the C code.

[5] Mutter is a window manager for X11 and Wayland compositor created by merging the previous *Metacity* window manager and Clutter graphic libraries

[6] Vala is high-level language that borrows its syntax from C#, but is compiled into C code with GObject and then to native code with a classic C compiler[9]. It heavily relies in GObject and related libraries, and uses automatic reference counting for memory management[10]

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

- Not all devices have support for RUST, and some probably will never have it.

For these cases where C is still a better option, but where the safety of the RUST Memory Ownership Model is desirable, is why we created CRUST. This paper presents this tool, its internal design and how it works.

1. Memory ownership model

As explained above, the main attractive in Rust is its memory ownership model and the checks implemented in the compiler to enforce it. This model allows to create safe applications, making it very hard to have *memory leaks* or *dangling pointers* in them. Also, it does this without using any kind of runtime (neither a *garbage collector*, nor other systems like *reference counting*), which gives it an unprecedented combination of security and performance.

Our new memory ownership model mimics the Rust one by using the basic assumption that a function has full ownership of every memory block ("*managed* block") it allocates, and is fully responsible for freeing them. It can also access a block only while it is being owned by it and hasn't been freed. Thus, if a block is allocated by a function, there are three possible courses of action:

- It can be freed inside the same function.
- It can be returned to the function caller, passing on to it both the ownership and the responsibility for freeing the memory block.
- It can be passed on to another function, so the caller function will loose not only the ownership and the responsibility for freeing it, but also the rights to access that block (because the block must be freed by the called function before returning to the caller).

Although, in theory, these three rules are enough to guarantee safety, in practice, to make coding easier, the concept of *borrowing* an allocated block is

introduced: when a function receives a borrowed block, that function *must not* free it, so it must not only avoid calling the *free()* function, but also not to pass that block to another function that could free it. This means that a borrowed block can only be borrowed to other functions and sub-functions.

Without borrowing, every function that didn't want to free a block would have to return it again to the caller, in order to also return the ownership. This would make the code clumsy.

Of course, it is not enough to just check that a memory block is freed or passed on once in a piece of code, but all the code execution paths must be considered. An example is when a memory block is passed on to another function inside an *if* statement: when the comparison is *true*, the block will be freed in the called function and the original function must not use it from that point onward; but when the comparison is *false* then the original function will still have the ownership, so it still can use the block and must free it or return it before exiting.

Finally, there is another rule to increases legibility and maintainability: each *managed* block can be referred only by a unique pointer; thus there can't be two pointers referring to the same block (although there is a way for relaxing this in cases that require it, as we will see).

2. Requirements for the implementation

When deciding how to implement this memory ownership model in C language in a generic and flexible way, there were several requirements that we considered unavoidable:

- The code must still be *pure C*, thus no code transformations are allowed. This means that solutions based on preprocessors (like Qt's *Meta-Object Compiler*[12]) that add extra functionality to the code were completely rejected.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

- For the same reason, no specific macros are allowed. The programmer must be able to retain full control over the code and never rely on any kind of code transformation that could have side effects.

This is because the idea is not to create a new programming language, but to just add the bare minimum required to pure C to allow a static analyzer to ensure that the code follows the specified memory ownership model. Also, it is desirable to avoid adding extra steps in the compilation and build sequence, thus allowing the programmer to use any toolchain, compiler system, IDE... that they already use. This way, the programmer will just run the tools periodically on the generated code to find bugs, but not every time they compile the code.

Also, since C language wasn't originally designed to use this model, it is not desirable to enforce it in each and every memory block, so there must be a way to specify which memory blocks must be managed and which ones shouldn't.

The solution was to define several new reserved words with the prefix *_crust*. These reserved words must be completely unnecessary for the compiler, and must be completely ignored by it. They must be meaningful only to the static analyzer. Thus, the code must be compiled as if these words didn't exist, but when using our static analyzer it must be able to interpret them to know whether a memory block is *managed* or *unmanaged*.

3. New reserved words

The reserved words can be divided into three groups:

• Qualifiers: these work like *tags* and just modify the meaning of a pointer variable or function parameter (in much a way as *const* or *volatile* from C), allowing to specify that the block pointed by it must be managed and in which way. These are:

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

- _crust_
- _crust_borrow_
- _crust_recycle_
- _crust_alias_
- _crust_not_null_
- _crust_override_

• Loop qualifiers: allow to give extra information about a *for* or *while* loop.

- _crust_no_0_

• Control commands: allow to control the way the static analysis is performed

- _crust_disable_
- _crust_enable_
- _crust_full_enable_
- _crust_set_null_
- _crust_set_not_null_

To ensure that the compiler ignores these words, the following header file must be added in all source files:

```
#ifndef ENABLE_CRUST_TAGS

#ifndef __crust__
#define __crust__
#endif

#ifndef __crust_borrow__
#define __crust_borrow__
#endif

#ifndef __crust_recycle__
#define __crust_recycle__
#endif

/* all other reserved words */
#endif
```

It ensures that all the reserved words will be defined as “blank” (and, thus, will be fully removed by the C preprocessor during normal compilation) when *ENABLE_CRUST_TAGS* is not defined. This is a must because the

static analyzer must pass the source code through the standard C preprocessor too, just like the compiler, to ensure that the macros used by the programmer will be correctly expanded to the desired code, but this time without removing the reserved words.

4. Working with CRUST

As mentioned in section III, the idea is to annotate what otherwise is just plain standard C code with some extra qualifiers that have meaning only for the static analyzer, but which will be discarded by the C preprocessor during normal compilation.

The main qualifier is *_crust_*. It allows to specify that a pointer refers to a *managed* block. Whenever a pointer is defined as *managed* it must be operated only in a *managed* way.

Of course, not every pointer has to be marked as *managed*; as a rule of thumb, any pointer to a memory block that won't be accessed through pointer arithmetic is a potential candidate to be defined as *managed*. Examples of these are:

• opaque pointers to library handlers, like the *FILE* pointer from *stdio*, the *xcb_connection_t* from XCB...

• any pointer to a *GObject*[7] object

• pointers to C structs.

But any pointer that will be modified with pointer arithmetic must not be declared as *managed*, because CRUST forbids it.

[7] GObject, or GLib Object System, is a C library that provides a portable object system in C. It was originally created as part of GTK[13], the Gimp ToolKit, but with the launch of GTK 2.0 it was removed from it and launched as part of the GLib library

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

As explained in section II, a *managed* block will be evaluated by the static analyzer to ensure that it is not used before being assigned a block or after being freed, and also that it is properly freed, passed on to a child function or returned before the end of the function.

The *_crust_* qualifier can also be used in a function definition to specify that a parameter and/or the return value is a *managed* block. In this case, trying to pass an *unmanaged* block as a parameter when the function expects a *managed* one, or a *managed* block where the function expects an *unmanaged* one, is an error. Trying to assign an *unmanaged* block into a *managed* pointer, or vice-versa, is also an error. This is shown in the following piece of code:

```
void a_function(unsigned int *);

void another_function() {
        __crust__ unsigned int * param;

        // ERROR: param is managed, but
        // the function expects an
        // unmanaged block

        a_function(param);
}
```

This is 100% legal C code, but when it is processed by CRUST it returns the error:

```
Analizing file
CRITICAL: Expected a non _crust_ variable as argument 1 when calling function 'function' at line 10, but passed a _crust_ variable
Total: 1 errors.
```

This happens because *param* is defined as a *managed* block at line 4, but the function doesn't expect a *managed* block. This is considered a *syntax error*, thus it can hide other errors. CRUST doesn't allow to do this because it can't determine what will happen to that block inside that function (this is: CRUST won't know if it will be freed or not).

CRUST also tracks the current status for each *managed* pointer, ensuring that they are used only after a block has been assigned. This is paramount because not taking care of this would result in *core dumps* during execution. This piece of code shows this:

```
void a_function(__crust__ unsigned int *);

void another_function() {
        __crust__ unsigned int * param;

        // ERROR: param is managed, but
        // is used before being assigned

        a_function(param);
}
```

Here, CRUST detects an error at line 9 because the pointer hasn't been previously assigned to a block, thus returning this error:

```
Analizing file
ERROR: Argument 1 when calling function 'function' at line 9 isn't initialized.
Total: 1 errors.
```

A *managed* block can be created using *malloc* and family, or from a factory function that returns an initialized *managed* block of the desired type[8]. Also, every *managed* block must be passed on to a function, returned to the function caller, or freed before the function ends in all possible execution paths.

[8]An example is an object's constructor function in GObject.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

```
typedef __crust__ void * managed_t;

managed_t factory();

void a_function(managed_t);

void another_function(int b) {

        managed_t param = factory();

        if (b < 0) {
                a_function(param);
        }
}
```

In this piece of code, a *typedef* is used to simplify the syntax: *managed_t* is a pointer to a *managed* block. After that, two functions are defined: one is a factory function that returns an initialized *managed* block, and the second one consumes a *managed* block. After being processed by CRUST it returns this error:

```
Analizing file
ERROR: Memory block 'param', initialized at line 9, is still in use at exit point in line 14
Total: 1 errors.
```

This happens because CRUST detects that there is an execution path, when *b* is equal or greater than zero, where the *managed* block pointed by *param* is not freed before the function ends. It is, thus, mandatory to ensure that *param* is passed on to a function that expects a *managed* block also in the second path, in order to free it before exiting the current function.

CRUST can detect some of the same errors detected by the compiler, but not all, so it is mandatory to ensure that the code doesn't produce warnings or errors when compiled before trying to pass it to CRUST. Here is an example:

```
typedef __crust__ struct str1 * str1_t;
typedef __crust__ struct str2 * str2_t;

str1_t factory_1();
str2_t factory_2();

void a_function(str2_t);

void another_function() {

        str1_t param_1 = factory_1();
        str1_t param_2 = factory_2();

        a_function(param_1);
        a_function(param_2);
}
```

This code doesn't return errors after being processed by CRUST, but it returns several warnings when compiled because the pointer types passed on to each function are incorrect in lines 12, 14 and 15. This piece of code also shows that, in the same code, can be *managed* blocks of several types, but they are not interchangeable.

Of course, CRUST also ensures that a pointer to a *managed* block is not overwritten before being freed, as shown in this piece of code:

```
typedef __crust__ void * managed_t;

managed_t factory();

void a_function(managed_t);

void another_function(int b) {

        managed_t param = factory();

        if (b < 0) {
                a_function(param);
        }

        param = factory();
        a_function(param);
        param = factory();
        a_function(param);
}
```

When CRUST processes this code it returns the following error message:

```
Analizing file
ERROR: Assignment to 'param' at line 15, which was already assigned at line 9
Total: 1 errors.
```

When assigning a block to *param* at line 15, in one of the possible execution paths (when *b* is equal or greater than zero) the old block has not been freed. This is not allowed because it would result in a *memory leak*, and this is why CRUST complains. But in line 17 it is possible to assign a new *managed* block in the same pointer because there is no doubt that it has been freed before, in line 16.

The possible states for a pointer are not only *unassigned*, *freed* and *assigned*; the static analyzer tracks if a variable or pointer is or can be NULL; this is why this code does not produce errors when checked by CRUST:

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

```
typedef __crust__ void * managed_t;

managed_t factory();
void a_function(managed_t);

void another_function(int b) {

        managed_t param = factory();
        if (param != NULL) {
                a_function(param);
        } else {
                param = factory();
                a_function(param);
        }
}
```

The *factory* function returns either a pointer to a block, or NULL. If the pointer is not NULL, the block will be freed during the call to *a_function*; but if it is NULL, then the assignment at line 14 will be legal because there is no memory block that can leak.

A. Borrowing blocks

It is not always desirable to give the ownership of a *managed* block to the called function, because the caller may still want to use it after the call. The *naive* solution for this case is to ensure that the called function returns the same block that is passed on as a parameter, but it is not a perfect solution because it not only makes the code clumsy, but it also can't be used when more than one block must be preserved.

To avoid this problem, both Rust and CRUST allow to *borrow* a *managed* block. When a function accepts a parameter as *borrowed*, the caller can be sure that it won't be freed, thus retaining the ownership after the call. This can be seen in this piece of code:

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

```
typedef __crust__ void * mnged_t;

void afunction(__crust_borrow__ mnged_t);

void another_function(mnged_t param) {

        afunction(param);
}
```

Analizing file
ERROR: Memory block 'param', initialized at line 5, is still in use at exit point in line 8
Total: 1 errors.

Since the *managed* block pointed by *param* is *borrowed* to the function *a function*, it isn't freed after returning, so the calling function still has the responsibility of managing it. That is why CRUST complains.

Also, a *borrowed* block can be passed on only as a *borrowed* parameter to subfunctions, because it must never be freed by the called function, as seen in this piece of code:

```
typedef __crust__ void * mnged_t;

void a_function(mnged_t);

void another_function(
        __crust_borrow__ mnged_t param) {

        a_function(param);
}
```

Analizing file
CRITICAL: Argument 'param' at position 1 when calling function 'a_function' at line 8 is borrowed, but is used as a
non-borrow argument
Total: 1 errors.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

Since *a_function* receives a *managed* block, it assumes that it must free it, thus it is an error to pass on a *borrowed* block, because when a block is *borrowed* it must not be freed by the called function, nor by any of the functions called by it.

B. Not null parameters

By default, when CRUST analyzes the code it presumes that any pointer parameter passed on to a function, or any return value from a function, can be NULL or NOT NULL (of course, until an *if* statement checks whether a pointer is NULL or NOT NULL). But it is possible to annotate the code to specify that a parameter passed on to a function must never be NULL, or that the return value from a function will never be NULL.

```
typedef __crust__ void * mnged_t;

void a_function(
        __crust_not_null__ mnged_t);

void another_function(mnged_t param) {

        a_function(param);
}
```

```
Analizing file
WARNING: Argument 'param' at position 1 when calling function 'a_function' at line 7 is defined as not_null, but is being called with a possible NULL value
Total: 1 errors.
```

Here *param* is the parameter received by *another_function*, which can be NULL or NOT NULL, thus is an error to pass it on as-is to *a function*, because the parameter must be a NOT NULL pointer. A solution could be to put the call inside an *if (param != NULL)* statement, which will be detected by CRUST and avoid the error:

```
typedef __crust__ void * mnged_t;

void a_function(
        __crust_not_null__ mnged_t);

void another_function(mnged_t param) {

        if (param != NULL) {
                a_function(param);
        }
}
```

This code will not produce errors because if *param* is NOT NULL, it will call *a_function* and the block will be freed; but if it is NULL, nothing will be called (this also will not generate an error because CRUST knows that if a pointer is NULL there is no block, so there is no risk of a *memory leak*).

Of course, when a parameter in a function is labeled as *not_null*, when CRUST analyzes the execution paths it will presume that the pointer is never NULL.

It is also possible to label a return value as *not_null*. In this case, when analyzing the execution paths, CRUST will ensure that the return value is never NULL.

C. Recycling managed blocks

This qualifier was added only as a helper for the first project, and probably has no use in others. It can be applied only to up to one *not borrowed managed* pointer in a function's parameters, and that function must also return a *managed* block. In that case, it means that if the *recycled* parameter is NOT NULL, then the returned value will also be NOT NULL; but if the parameter is NULL, the returned value can be NULL or NOT NULL.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

This is used in some functions that, internally, recycle the memory block previously allocated for a *managed* block to create a new *managed* block, instead of freeing the memory first and then asking for a new allocation; but that when passed a NULL pointer instead, the function allocates a new memory block.

D. Pointer aliases

By default, each *managed* block can be assigned only to a single pointer. Trying to assign the same block to several pointers will only pass the ownership from one to the next, while all the previous pointers will end in a *freed* status. This is on purpose, because having several pointers to the same *managed* block messes up the code and makes it harder to maintain and understand.

Unfortunately, there is a common case where it is useful to have several pointers to the same block: when walking through a linked list. In this case it is common to have a *for* loop and a temporary pointer that is used to walk through the linked list. To allow to implement this, CRUST allows to create an *alias* pointer, a specific type of pointer to a *managed* block that can, if needed, point to a block that is already pointed by another reference. This piece of code shows how to use it:

```
struct block {
        void *data;
        struct block *next;
}

typedef __crust__ struct block * mnged_t;

void a_function(mnged_t a_list) {

        __crust_alias__ mnged_t tmp;

        for(tmp = a_list;
            tmp != NULL;
            tmp = tmp->next) {
                // do things
        }
        // more things
}
```

Here, the *tmp* pointer is an alias, so it can point to the same block than *a_list*. Thus *tmp* and *a_list* become "*entangled*". When this happens, passing any of them on as a *not borrowed* parameter will mark BOTH as freed. But if a new block is assigned to the alias when both pointers are still *entangled* (like in line 14 with the code *tmp = tmp-¿next*), both pointers become *disentangled*, so passing one on as a *not borrowed* parameter will mark only that pointer as freed, and the other will retain its current state.

Although it is possible to use aliases in other cases, it is strongly discouraged, because *aliases* break the assumption that each *managed* block is referred only by an unique pointer and make the code harder to understand and maintain.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

E. Overriding function definitions

When using external libraries in a project with CRUST, the function declarations in the header files have to be modified to add the needed CRUST qualifiers. This has several problems:

- Usually, the programmer lacks *root* access to modify the files *system-wide*.
- It is uncommon to use all of the function available in a library, usually requiring only a handful of them, so it is not a must to modify all the function definitions in the header files.
- Changes in the header files may be overwritten every time the library is updated by the operating system's package manager.

The naive solution to these problems is to copy the library header files into the project folder, modify only the functions used in the code, and use them to compile it. This solves the first two items, but leaves the third unresolved: every time the library is updated, the programmer is required to find and append the differences in the header files.

A more elaborated solution would be to still include the original headers and add another header file with only the definitions that must be changed. Unfortunately, this won't work: during compilation there will be no problems because CRUST qualifiers are removed by the C preprocessor, so both definitions will match; but when using CRUST, it will find two different definitions for each function, one with the qualifiers and another without them, and it will raise an error.

The solution is to use the qualifier *_crust_override_* with the modified definitions. When CRUST finds it, it will give priority to that definition over the ones without the *override* qualifier.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

```
__crust_override__ xcb_connection_t
    __crust__ *xcb_connect(
            const char *displayname,
            int *screenp);

__crust_override__
    void *xcb_get_property_value(const
        xcb_get_property_reply_t
            __crust_borrow__ *R);
```

In this example, the function definition for *xcb_connect* is replaced by one that returns a *managed* block, which is the *xcb* connection. Also the function definition for *xcb_get_property* is also replaced by one that receives a *borrowed* block with a connection to the X server.

F. Loop qualifiers

The qualifier *crust_no_0* allows to specify that a loop runs at least once. To understand this better let's see how CRUST evaluates loops:

Evaluating a loop takes into account that it can be run zero times, once, or more than once, depending on the condition. This is why *for* and *while* loops are analyzed in three cases:

1. when the loop is not run at all: it evaluates when the condition isn't met right at the beginning.
2. when the loop is run once and the loop exits.
3. when the loop is run twice: it allows to detect errors when the loop is run several times before exit.

Checking these three cases allows to detect common errors like writing code that fulfills the rules when it is run once or more times but not when it is run zero times, or that honors them when it is run zero or one times but not when it is run

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

more than once. Of course, a *do...while* loop will be tested only in the two latter cases because it always runs at least once.

Now the usefulness of this qualifier becomes clear: it is very uncommon for a *for* loop to run zero times, but the static analyzer can't know that, so in order to avoid false errors from CRUST it is possible to tag a *for* or *while* loop with *_crust_no_0_*. This will instruct CRUST to check only the cases where the loop runs one or more times, but not zero times. Of course, this qualifier is not allowed in *do...while* loops because they always run at least once.

G. Control commands

Sometimes, there is no way of doing something without breaking the rules. In these cases it is possible to disable the error detection for specific blocks of code. To manage this, CRUST has an internal variable named *disable counter*, which is managed with these commands:

- *_crust_disable_* increments the *disable counter*
- *_crust_enable_* decrements the *disable counter*
- *_crust full enable* sets to zero the *disable counter*

When *disable counter* is zero, CRUST will detect and show warnings and errors in the code as expected; but when it is non-zero, CRUST will ignore them. This allows to nest the *disable* and *enable* commands, since as many *enables* as *disables* are required to re-enable the error detection. This can be overriden with *full_enable*, which forces the error detection again no matter how many *disables* have been executed previously.

H. Global pointers

Working with *managed* blocks pointed by global variables is not trivial, because it implies many cases that a static analyzer can't detect. This is why the

rules must be relaxed in this case.

Global pointers to *managed* blocks are allowed, but by default, when entering a function, they are considered to be always assigned; that is, their value can be NULL or NOT NULL, so it is mandatory to check the state with an *if* statement before trying to assign a block or to use it as a parameter.

But since it is common that, in some functions, the true state of a global variable is known due to the logic of the program, there are two statements to specify this and avoid putting an unnecessary *if*. These are *__crust_set_null_()* and *__crust_set_not_null_()*:

```
typedef __crust__ void *crust_t;

crust_t global_var;

void a_function() {

    // global_var is never NULL
    __crust_set_not_null__(global_var);

    // process global var and free it
    // ...

    enable_interrupt();
}

void interrupt_handler() {

    // global var is always NULL

    __crust_set_null__(global_var);

    global_var = ...

    trigger_fnc(a_function);

}
```

This example shows this case: there is an interrupt handler that stores a block in a global variable and triggers a function that, in user-mode, will read that block and enable the interrupts again. Every time the user-mode function is triggered, the global variable is known to point to a *managed* block, because the function is called only after an interrupt; also, whenever the interrupt handler is called, the global variable is known to be *empty* because it is assigned only in the interrupt handler, and it can't be called again until the user-mode function frees the global variable and enables again the interrupts.

There are several differences between a local and a global variable. A block can be assigned simultaneously to a local and a global variable, but if the block is freed before exiting the function, it is mandatory to set the global variable to NULL. Not doing it is an error, and CRUST will report it. This can happen either if the local or the global variable is passed on to a function that frees it. Also, assigning the same block to more than one global variable is not allowed.

Of course, setting the status for a global variable should be done always before accessing it for the first time. CRUST will detect any attempt to set it after having accessed the variable and will report it with a *warning*.

5. Implementation details

The static analyzer is divided into two main sections: the C parser, written as a classic *LEX/YACC* parser and compiled as a *shared library* into native code; and the *execution path evaluator*, written in *Python 3*.

A. C parser

The C parser was created using the *LEX/YACC* rules written originally by Jeff Lee and published by Tom Stockfisch in Usenet, and currently maintained by Jutta Degener in her personal webpage[14] [15]. This C99 parser has been

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

expanded with several GNU extensions. The following extensions are recognized at syntax level but completely ignored during the static analysis:

- #pragma
- _builtin va list
- _signed
- _extension
- _prog
- _restrict
- _inline
- _attribute (...)
- _asm [XXXX] (...)
- asm [XXXXX] (...)
- _alignof (...)
- _typeof (...)
- _builtin offsetof(...)

Also, support for concatenated strings (in the form ”a concat””enated string”), statements inside parentheses and ellipsis in CASE statements (like CASE n1 ... n5) has been added.

The C parser receives the source code after being managed by the *C preprocessor*, and generates an *AST tree* that is passed on to the main program. The *AST tree* is quite unusual because it looks more like a linear list of nodes than a proper tree, as seen in figure 1. This format greatly simplifies the execution of loops and conditional statements because a branch can be easily evaluated just by duplicating the list at the branching point and removing only the unnecessary statements in each duplicate. This will be explained in depth later.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

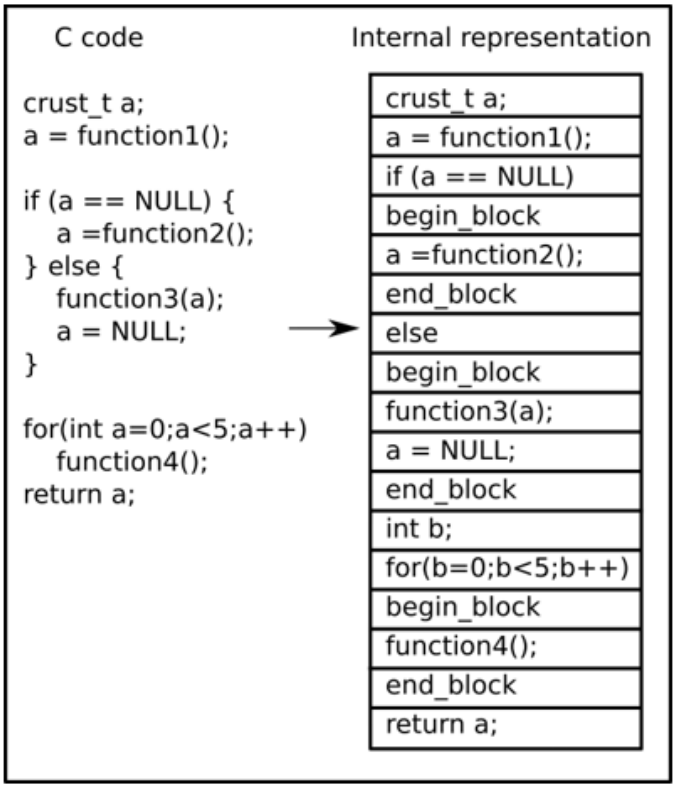

Figure 1: A piece of C code with a branch is translated from the *original C code* into the *internal representation* generated by the *LEX/YACC* parser. Code blocks inside *if*, *switch*, *for* loops, *while* loops and *do...while* loops are surrounded by *begin_block* and *end_block* statements to allow, in further steps, to separate all the execution paths.

B. The execution path evaluator

The *execution path evaluator* is the heart of the static analyzer. It does all the hard work.

The first step is to take the *AST tree* created by the parser and build a list with all the global variables and functions in the code. Then, these functions are processed one by one.

Whenever a function is going to be analyzed, a new *execution environment* is created. It consists of a list with all the remaining code to be "executed" and the

current variables (global, local, and parameters). The execution environment holds the type (whether it is a *managed* pointer or not, and if it is *borrowed*), and the current status for all the variables. The status of a variable can be one of the following:

- *UNINITIALIZED*
- *NULL*
- *NOT NULL*
- *NULL OR NOT NULL*
- *FREED*
- *FREED OR NULL*

By default, all parameter variables are set as *NULL_OR_NOT_NULL* (meaning that it is unknown if they point to a *managed* block or are just NULL) unless they are defined with the qualifier *_crust_not_null_*, in which case the status will be set to *NOT_NULL*.

Global *crust pointers*, by default, are set as *NULL_OR_NOT_NULL*, but if it is known that a global one is positively *NULL* or *NOT_NULL*, it is possible to specify it by using *_crust_set_null_(variable_name)* or *_crust_set_not_null_(variable name)*, as explained in section V-H.

Local variables are set to *UNINITIALIZED* when created, and they keep that status until they are assigned a value (which can be the value of another pointer, the NULL value, or the return value of a function).

C. Processing linear code

The main actions that form a piece of *linear code* and that matter to CRUST are assignments, reads, and calls that involve *crust pointers*. Other assignments, reads and calls aren't evaluated unless necessary. The *execution path evaluator* will take each instruction in the piece of code and check if a *crust pointer* is

involved in any of the steps needed to run it. If that is the case, it will simulate its behavior.

If the statement creates a new variable, it will be added to the execution environment and its status will be set to *UNINITIALIZED*.

When a *crust pointer* is used inside a statement, it will be set to *FREED* status if its previous status was *NOT_NULL*, or *FREED_OR_NULL* status if its previous status was *NOT_NULL_OR_NULL*, unless it is passed as a *borrowed* block or when it is used in a comparison (like in *if (variable == NULL)* comparison). Trying to pass on a *borrowed crust pointer* as a *non-borrowed* parameter is an error, because the called function will free it, and CRUST will report it.

When a statement assigns a value to a *crust pointer*, first it is checked whether its status is *UNINITIALIZED*, *NULL*, *FREED_OR_NULL* or *FREED*; if the status is none of the above, it means that the code is trying to overwrite a *managed* block, which would cause a memory leak if allowed, so CRUST will show an error message with the line number. If there is no error, the value to be assigned will be evaluated as the new status; if it is the return value of a function, it usually will mean *NULL_OR_NOT_NULL*, unless the return value is defined as *crust_not_null_*. If it is another *crust pointer*, the current status of the origin pointer will be copied and the original pointer will be set to *FREED* or to *FREED_OR_NULL*, depending on the previous state.

If a statement tries to read from a *crust pointer*, or pass on a *crust pointer* as a function parameter, and the *crust pointer* is still *UNINITIALIZED*, it will be considered a breach in the memory management rules and an error will be displayed with the corresponding line in the code where the error is located. The same happens if the status is *FREED*, because that means that the code is trying to pass on a *dangling pointer*.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

When the function reaches the end, the status of all the *crust pointers* will be checked to ensure that the rules have been honored:

- If a local variable or a *non-borrowed* parameter is *NOT NULL* or *NULL OR NOT NULL*, it will be considered an error because it means that there is a memory leak.
- Global *crust pointers* must be *NULL*, *NOT_NULL* or *NULL OR NOT NULL*, but never *FREED* nor *FREED OR NULL*, because that means that a global variable contains a *dangling pointer*. Thus, every time a global *crust pointer* is used (and, thus, freed) it must be set manually to NULL to ensure that other functions won't find a *dangling pointer* in it.
- When the returned value is a pointer to a *managed* block, it must be *NULL*, *NOT NULL* or *NULL OR NOT NULL* status, but never one in *UNINITIALIZED*, *FREED* or *FREED OR NULL*, because that would lead to a *dan- gling pointer*.

D. Branches

Conditional statements and loops have in common that they contain at least one branch: the code breaks its linear nature and branches out in ways that may or not be followed in an actual execution. However, in the case of a static analyzer, it must follow all the branches to ensure that there aren't any errors in any path.

With *if*, *switch* and *condition? A : B* statements this is an easy task because the number of paths is finite and known in advance, so the solution used is to duplicate the remaining code in as many branches as needed, remove in each one the unused code, set the tested variables to the right values, and evaluate each one independently. This can be seen in figure 2, where an *if* statement with an *or-ed* condition is evaluated.

for, *while* and *do...while* loops are more complex because, except for trivial cases, it is not possible to know in advance how many iterations they will take and, thus, how many branches will be needed. Fortunately, it is not necesary to

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

evaluate all of them, but only three branches in the case of *for* and *while* loops, and two for *do...while* loops. As explained in section V-F, the first branch evaluates the case when the inner code in the loop is never run because the condition is false before starting (obviously, this branch is never created in the *do...while* loops because these always run at least once, nor in *for* or *while* loops labeled with *crust_no_0_*). The second branch evaluates the inner code when the loop runs exactly once. Finally, the third branch evaluates the case when the loop runs twice. It is not necesary to evaluate more cases because the third branch allows to detect all possible errors when the inner code is run more than once. This is shown in figure 3.

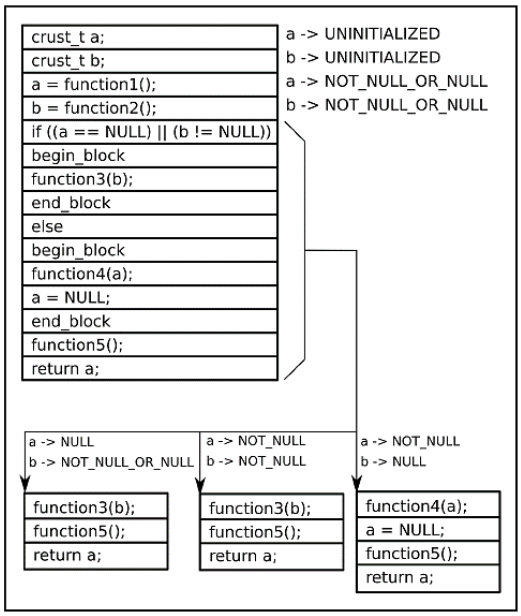

Figure 2: The code path, already transformed in the internal representation, is *run* to create the variables and assign its new status. When the execution reaches the *if* statement, the current internal representation and the list of variables are cloned, the blocks that don't belong to each branch are removed, the status of the variables is set in each path according to the condition tested, and each branch is *run* independently. This example also shows how the static analyzer follows the same *short- circuit* rules as the ANSI C compiler to reduce the number of branches from four to three, by evaluating a condition as *true* or *false* as soon as possible; in this case, when *a* is *NULL*, the value of *b* doesn't matter.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

Finally, the *goto* branches are the most problematic ones. Historically they have been considered *harmful*[16], although in recent years they are being reconsidered under several circumstances, like in the Linux kernel: Linus Torvalds and other programmers consider that a judicious use of the *goto* statement can lead to a cleaner code than a pure structured approach[17].

The biggest problem with *goto* is that it is an *statement* that gives too much freedom to the programmer, allowing to create complex loops and behaviors that can defeat the most clever static analyzer[18]. But at the same time it has its legal use cases, so forbidding it completely is a bad idea. Fortunately, the main use case for *legal gotos* is to exit from an inner loop without needing extra flags to also exit outer loops, and in these cases they always jump into a label located *after* the *goto* statement. This case has the advantage that it is impossible to have an infinite loop (which is the main risk when jumping to a point located before the *goto* statement), so it can be managed as easily as a *break* statement.

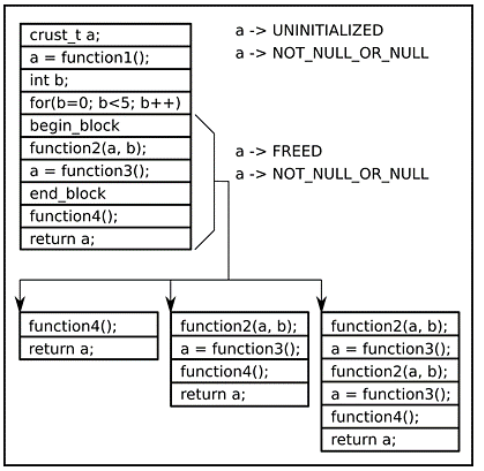

Figure 3: An example of a *for* loop and how it is evaluated to ensure that there are neither *memory leaks* nor *dangling pointers*. Three branches are evaluated, one where the loop never happens, another where the loop runs once, and finally another one where the loop runs twice, which is enough to detect if there are errors when the loop runs more than once.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

E. Speed optimization

The model described works fine and is extremely useful. Unfortunately, the original, naive implementation proved to be inadequate with some common pieces of code. The problem is that having several loops and branches in the same function can result in a combinatorial explosion. An example is shown in figure 4.

That simple code, with six *if* and three *for* statements, expands into 1728 possible execution paths (each *if* statement multiplies by 2 the number of paths, and each *for*, by 3). Adding *AND* or *OR* statements to the conditions in the *if* statements would make that number even bigger. A function in our project had several nested *if* and *for* statements, resulting in more than six million paths to evaluate. These big numbers taxed the original static analyzer so much that it needed more than 36 hours to evaluate all of them.

To solve this, a solution was developed that discarded paths identical to other paths that have already been evaluated. To do so, each time the analyzer reaches an *end_block* statement in the *internal representation* (which corresponds to the end of an inner block in the code, usually to a closing brace), it will compare the current status of every variable in the current path with the status in the same code point in other paths that already have executed it. If all statuses are coincident, this path is identical to the previously executed path and may be discarded. But if there are differences, the status from the current path is stored along with the others to allow to compare it with future execution paths.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

```
typedef __crust__ void * mnged_t;

mnged_t afunction();

void function2(
        __crust_borrowed__ mnged_t);

void function3(mnged_t);

void another_function(mnged_t param1,
                      mnged_t param2,
                      mnged_t param3) {

    if (param1 == NULL)
        param1 = afunction();

    if (param2 == NULL)
        param2 = afunction();

    if (param3 == NULL)
        param3 = afunction();

    for(int i=0; i<5; i++)
        function2(param1);

    for(int j=0; i<8; i++)
        function2(param2);

    for(int j=0; i<3; i++)
        function2(param3);

    if (param1 != NULL)
        function3(param1);

    if (param2 != NULL)
        function3(param2);

    if (param3 != NULL)
        function3(param3);
}
```

Figure 4: This piece of code is problematic because there are 1728 possible execution paths, which will require too much time to evaluate.

Thanks to this optimization, the time needed to analyze the problematic code went down to less than fifty seconds.

Ceres. Revista de Ingeniería, Tecnología, Ciencias Agropecuarias y Desarrollo Sostenible
ISSN: 3101-4895 / Vigo, Provincia de Pontevedra – España

F. Unitary tests

To guarantee that changes in the code won't break the analyzer, a set of unitary tests are included to ensure that it always performs as expected. Currently there are 242 unitary tests that check every rule and corner case found during the development. Also, these tests are an excellent source of examples for what to do and what not to do.

6. Future work

Although the current static analyzer supports several GNU extensions, these are insufficient to analyze code from the Linux kernel, or the internal code in GObject objects. To fix this, several improvements in the *LEX/YACC* parser are needed.

Another important change considered is to extend the supported language standard from the current ANSI C99 syntax to ANSI C11, to ensure that all the C standard code can be compiled.

Conclusions

In this work, we have developed a useful tool for programmers that allows them to improve their efficiency by helping to find *memory leaks* and *dangling pointers* in their C code, without having to learn and master a new programming language. The tool is fast and unobtrusive, and can be easily integrated in any workflow.

References

[1] N. D. Matsakis and F. S. Klock, II, "The rust language," *Ada Lett.*, vol. 34, no. 3, pp. 103–104, Oct. 2014.

[2] "Mozilla foundation." [Online]. https://foundation.mozilla.org

[3] "Rust foundation." [Online]. https://rustfoundation.org

[4] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin, "Experience report: Developing the servo web browser engine using rust," *CoRR*, vol. abs/1505.07383, 2015. [Online]. http://arxiv.org/abs/1505.07383

[5] "Redox project." [Online]. https://www.redox-os.org/

[6] "Thoughts on dx: Gnome and rust." [On- line]. https://siliconislandblog.wordpress.com/2016/10/31/ thoughts-on-dx-gnome-and-rust/

[7] "Federico mena section about rust." [Online]. https://people.gnome.org/~federico/blog/tag/rust.html

[8] "Original mail from miguel de icaza about the gnome project." [Online]. https://mail.gnome.org/archives/gtk-list/ 1997-August/msg00123.html

[9] "Vala - compiler using the gobject type system." [Online]. https://wiki.gnome.org/Projects/Vala

[10] "Vala's memory management explained." [Online]. https://wiki.gnome.org/Projects/Vala/ReferenceHandling

[11] "Initial posts about librsvg's c to rust conversion."

[12] J. Thelin, *Foundations of Qt Development*, J. Gilmore, Ed. Apress, 2007.

[13] "Gtk, the gimp toolkit official website." [Online]. https://www.gtk.org

[14] "Jutta degener ansi c99 grammar, lex specification." [Online]. http://www.quut.com/c/ANSI-C-grammar-l-1999.html

[15] "Jutta degener ansi c99 grammar, yacc specification." [Online]. http://www.quut.com/c/ANSI-C-grammar-y-1999.html

[16] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968

[17] "Linux kernel mailing list discussion about gotos in linux source code," 2003. [Online]. http://koblents.com/Ches/Links/ Month-Mar-2013/20-Using-Goto-in-Linux-Kernel-Code/

[18] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," Proceedings of the London Mathematical Society, vol. s2-42, no. 1, pp. 230–265, 1937

Declaration of conflict of interest and originality

In accordance with the *Code of Ethics and Best Practices* published in the *Ceres Journal*, the author *Costas Rodríguez, Sergio* declares to the Editorial Committee that he has no situations that represent a real, potential, or evident conflict of interest of an academic, financial, intellectual, or intellectual property nature related to the content of the article: *Adding a Rust-like memory ownership model to classic C language*, in relation to its publication. Likewise, he declares that the work is original, has not been published partially or entirely in any other medium, and that no ideas, formulations, quotations, or illustrations from various sources were used without clearly and strictly mentioning their origin and without being properly referenced in the corresponding bibliography. He consents to the Editorial Committee applying any plagiarism detection system to verify its originality.

The author declares that he did not use generative artificial intelligence tools for writing texts or interpreting data in the preparation of this manuscript.