C-rusted: The Advantages of Rust, in C, without the Disadvantages (Extended Abstract)

Roberto Bagnara
BUGSENG & University of Parma
Parma, Italy
Email: *name.surname*@unipr.it

Abramo Bagnara
BUGSENG
Parma, Italy
Email: *name.surname*@bugseng.com

Federico Serafini
BUGSENG & University of Parma
Parma, Italy
Email: *name.surname*@bugseng.com

I. The Problem

The C programming language is a crucial foundation of all current applications of information technology. It is, by far, the most used language when access to hardware is essential, even for critical safety-related and/or security-related systems.

There are very strong economic reasons behind the use of the C programming language, namely:

1) C compilers exist for almost any processor;
2) C compiled code is very efficient and without hidden costs;
3) C allows writing compact code thanks to the many built-in operators and the limited verbosity of its constructs;
4) C is defined by an ISO standard [1];
5) C, possibly with extensions, allows easy access to the hardware;
6) C has a long history of use, including in critical systems;
7) C is widely supported by all sorts of tools.

In fact, the C programming language is so widespread that it has no equals as far as the following criteria are considered:

- number of developers in low-level, safety-related and security-related industry sectors;
- number of qualified tools for compilation, analysis, testing, coverage, documentation, code generation and any other code manipulation;
- number and range of supported architectures.

On the other hand, several of C's strong points have negative counterparts, e.g.:

1) The fact that C code can efficiently be compiled to machine code for almost any architecture is due to the fact that, whenever this is possible and convenient, high-level constructs are mapped directly to a few machine instructions: given that instructions sets differ from one architecture to the other, this is why the behavior of C programs is not fully defined.
2) The reason why the maximum execution time of C programs can be estimated with good precision by expert programmers is because there is nothing happening under the hood and, in particular, there is no built-in run-time error detection.
3) The reason why C allows writing terse programs is the same reason why C code that is (intentionally or unintentionally) obscure is so common.

These negative sides of C compound when memory handling is concerned, as memory handling is fully under the programmers' responsibility:

1) memory references in C are (unless special care is taken) raw pointers that bring with themselves no information about the associated memory block or its intended use;
2) no run-time checks are made to ensure the safety of pointer arithmetic, memory accesses, and memory deallocation;
3) code involving memory addressing with pointers can be particularly opaque to peer review.

Some of the most common C memory issues are:

- dereferencing invalid pointers, including null pointers, dangling pointers (pointers to deallocated memory blocks), and misaligned pointers;
- use of uninitialized memory;
- memory leaks;
- invalid deallocation (including double free and free with invalid argument);
- buffer overflow.

Even though various coding standards (with MISRA C being the most authoritative one) and lots of "bug finders" exist, there is no verification tool that can *guarantee*, in a strong sense, the absence of a large class of software defects in a consistent, effective and repeatable way. In fact:

- MISRA C [2] provides guidelines for writing software that is *on average* much safer;
- bug finders find *some* recognizable instances of possible defects;
- systems based on deductive methods, like Frama-C [3], require programmers that are highly skilled in mathematical logic and, even when such programmers are available,

development time is multiplied by a factor from 2 to 4;[1]
- deep semantic analysis based on abstract interpretation only covers a small set of program properties and is affected by non-repeatable results due to the heuristics that are used to throttle the computational complexity of the analysis.

Of course, all this is not new. C criticism for the facility with which memory handling programming mistakes are committed date back to shortly after the language was made available to the public [4]. However, apparently the measure is full if the idea of rewriting (parts of) Linux in Rust —where committing such mistakes is significantly more difficult— is being taken seriously [5], [6], [7], [8], [9], [10], [11], [12]. It is not yet clear whether a global move to Rust is possible or even desirable. The main issues are:

- Legacy: there is too much legacy code written in C; the costs and risks involved in rewriting existing code bases (a good part of which has a more-than-honorable operational history and may be in perfectly good shape) are enormous.
- Personnel: retraining millions of developers to Rust would take time and lots of resources.
- Portability: for many MCUs used in the development of embedded systems, no implementation of Rust currently exists.
- Tools: while all sort of tools are available for C, the same thing cannot be said for Rust.

So, what does the current ferment about Rust tell us? That the industry is ready to accept that programmers take a more disciplined approach by embracing *strong data typing* enhanced with *program annotations*. This is the real change of perspective: the technology to assist this new attitude in the creation of C code with unprecedented integrity guarantees is available, in its essence, since decades.

II. From C to C-rusted

In this paper, we propose an alternative approach as a reply to the temptation of abandoning C in favor of other languages offering stronger guarantees in terms of safety and security. The idea is to take some concepts that have been proven to work well in other languages, such as "the ownership model" of Rust, into C, in order to define *C-rusted*: a safe, secure and energy-efficient flavor of C. The key points of our approach are the following:

1) C-rusted is based on *annotations* with which the programmer can express:
   a) ownership, exclusivity and shareability of language, system and user-defined resources;
   b) properties of resources and the way they evolve during program execution;
   c) nominal types and nominal subtypes compatible with any standard C data type.
2) As far as the compiler is concerned, all C-rusted annotations are macros expanding to nothing (with the exception of global annotations, which expand to something that obeys ISO C syntax and is ignored by the compiler). The (partially) annotated C programs being fully compatible with all versions of ISO C, can be translated with unmodified versions of any compilation toolchain capable of processing ISO C code.
3) In contrast to (2), a static analyzer can interpret the annotations and validate the program: if the static analysis flags no error, then the annotations are provably coherent among themselves and with respect to annotated C code, in which case said annotated program is provably exempt from a large class of logic, security, and run-time errors.
4) It is important to note that it is not only the presence of annotations that expresses information: even the absence of annotations has a definite meaning that is checked by the static analyzer so that any possible oversight or inconsistency is flagged. This characteristic is used, for example, to "reverse" some dangerous defaults of the C programming language: whereas in C an object of type pointer can be a null pointer, in C-rusted a pointer can be null only if it is annotated as such.

As a result:

- Legacy code can be reused as-is: for code that is safety-critical, annotations can be added in order to obtain proofs of safety, but no rewriting is required, thereby avoiding all risks that this would entail.
- There is no need to retrain the developers, apart from those that, working on safety-critical components, would have to get familiarity with the annotations.
- Existing C compilers can be used without any change, thereby ensuring maximum portability.
- All sort of tools, in addition to compilers, can also be used without any change.

III. C-rusted at a Glance

Even though the C programming language is (for the sake of efficiency only) statically typed, types only define the internal representation of data and little more: types in C do not offer programmers a way of expressing non-trivial data properties that are bound to the program logic. For instance:

- an open file has the same type as a closed file;
- a resource or a transaction has the same type independently from its state;
- an exclusive reference and a shared reference to a resource are indistinguishable;
- an integer with special values that represent error conditions is indistinguishable from an ordinary integer.

In C-rusted all these differences can be expressed incrementally, resulting in increased documentation, readability and reusability of the code. Most importantly, this enables the *C-rusted Analyzer*, which is based on the *ECLAIR Software*

[1]The reported factor comes from the personal experience of the first author, who has been trained extensively in mathematical logic up to and including the Ph.D. level. The same author has also used Frama-C in computer science courses for several years and found that the learning process is particularly difficult for students at the bachelor's and master's level, despite the fact that Floyd-Hoare logic played a fundamental role in such courses.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

extern void process(char *string);

int foo(const char *fname, size_t bufsize) {
  int fd = open(fname, O_RDONLY);
  char *buf = (char *) malloc(bufsize);
  ++fd;
  ssize_t bytes = read(fd, buf, bufsize - 1U);
  buf[bytes] = '\0';
  process(buf);
  return 0;
}
```

Fig. 1. A C program compiling with no warnings with `gcc -c -std=c18 -Wall -Wextra -Wpedantic`

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

extern void process(char *string);

int foo(const char *fname, size_t bufsize) {
  int fd = open(fname, O_RDONLY);
  char *buf = (char *) malloc(bufsize);
  ++fd^{w1};
  ssize_t bytes = read(fd^{w2}, buf^{w3}, bufsize - 1U);
  buf[bytes]^{w4 w5} = '\0';
  process(buf^{w6});
  return 0^{w7 w8};
}
```

w_1: After receiving the return value of `open()`, `fd` contains a file descriptor or the erroneous value -1: `fd` cannot be incremented.
w_2, w_3, w_4, w_5: `fd` is not a valid file descriptor, `bytes` may be -1, `buf` may be `NULL`.
w_6: Does `process()` take ownership, i.e., can it or must it deallocate its argument?
w_7: The open file description possibly obtained from `open()` is leaked here.
w_8: The block of memory possibly pointed to by `buf` is leaked here.

Fig. 2. The *C-rusted Analyzer* gives several warnings on the same C program

Verification Platform, to verify correctness on any platform, with any architecture, and for each compiler.

Consider the program in Figure 1, which the GNU C compiler compiles without any warning even at a very high warning level. The program contains a lot of likely mistakes, including the meaningless —but in C perfectly valid— numerical increment of a file descriptor.

When given to the *C-rusted Analyzer* the very same program triggers several diagnostic messages, summarized in Figure 2, where the notation w_n decorating a program point means that the indicated applicable warning is given at that program point.

A much saner version of the program is depicted in Figure 3 and allows us making some observations: while `fd` is declared having type `int`, the value of `fd` has properties that change throughout the function body; similarly for `buf` and `bytes`. In other words, the C-rusted type system is able to track how properties of resources change depending on the considered program point. The large part of the result is obtained without any annotation at all, thanks to the fact that the C Standard Library and the POSIX Library have been annotated once and for all (and the same can be done with any frequently used library). Moreover, annotations are not heavy and do not clutter the code; type qualifiers, such as `e_hown()`, can also be embedded in typedefs and a proper choice of typedef names

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <crusted.h> // Include C-rusted declarations, e.g., for e_hown().

// The actual parameter must be a valid (hence, non-null) pointer
// to a char array in the heap of which process() will take ownership:
// the caller must have ownership for otherwise it would be unable to pass it on.
extern void process(char * e_hown() string);

int foo(const char *fname, unsigned bufsize) {
  int fd; // (The value of) `fd` is indeterminate.
  fd = open(fname, O_RDONLY);
  // `fd` is either the erroneous value -1 or an owning reference to an open file description.
  if (fd == -1)
    return 1;
  // `fd` is definitely an owning reference to an open file description.

  char *buf = (char *) malloc(bufsize);
  // `buf` is either NULL or an owning reference to a heap-allocated char array.
  if (buf == NULL || bufsize == 0U) {
    (void) close(fd);
    // Ownership of the open file description moved from the actual parameter
    // to the formal parameter of close(), which will close it:
    // no open file description leak; `fd` cannot be used anymore but it can be overwritten.
    return 1;
  }
  // `buf` is definitely an owning reference to a heap-allocated char array.

  ssize_t bytes = read(fd, buf, bufsize - 1U); // No ownership move, resources are borrowed.
  // `bytes` is either the erroneous value -1 or the number of bytes read into `buf`.
  if (bytes == -1) {
    free(buf);
    // Ownership of the heap-allocated memory moved from the actual parameter
    // to the formal parameter of free(), which will deallocate it:
    // no memory leak, `buf` cannot be used anymore but it can be overwritten.
    (void) close(fd); // Ownership moved from actual to formal parameter, as in line 22.
    return 1;
  }
  // `bytes` is definitely the number of bytes read into `buf`.
  buf[bytes] = '\0';

  process(buf);
  // Ownership of the heap-allocated memory moved from the actual parameter
  // to the formal parameter of process(), which will deallocate it:
  // no memory leak, `buf` cannot be used anymore but it can be overwritten.

  if (close(fd) != 0) // Ownership moved from actual to formal parameter, as in line 22.
    return 1;

  return 0;
}
```

Fig. 3. The *C-rusted Analyzer* gives no warning on this version

also helps readability and understandability.

IV. References

The annotation language allows expressing constraints on the use of *resources* via *references*. A *resource* is anything that a C program has to manage. Generally speaking, resources need to be allocated or reserved, need to be manipulated by operations that have to be performed in some predefined ordering, and need to be destroyed or deallocated or unreserved. C-rusted supports different kinds of resources: memory resources and any language-defined, system-defined or user-defined abstraction with a definite lifecycle. A *reference* is any C expression that is able to refer to a resource. A valid pointer and a file descriptor are example of references.

A. Owning, Exclusive and Shared References

C-rusted distinguishes between different kinds of references:

Owning references: An *owning reference* to a resource has a special association with it. In a safe C-rusted program,[2] every resource subject to dynamic release (as opposed to automatic release, as in the case of stack variables going out of scope), must be associated to one and only one owning reference. Through the program evolution, the owner of a resource might change, due to a mechanism called *ownership move*, but at any given time said resource will have exactly one owner. The association between the owner and the owned resource only ends when a designated release function is called using the owner as parameter. Note that an owning reference is a kind of *exclusive reference*.

Exclusive references: As the name suggests, an *exclusive reference* grants exclusive access to a resource and, as such, both read and write operations are allowed through the exclusive reference. As a consequence of this fact, no more than one usable exclusive reference to the same resource can exist at any given time; moreover, the existence of a usable exclusive reference is incompatible with the existence of any other usable reference to the same resource. Note that the referred resource cannot be released via an exclusive non-owning reference (only an owning reference allows that).

Shared references: A *shared reference* to a resource can be used to access the resource without modifying it. As read-only access via multiple references is well defined, there may exist several usable shared references to a single resource. However, during the existence of a shared reference, no exclusive references to the same resource can be used.

B. Optionality

In C-rusted an *optional type* is any type having a subset of its values that are "reserved" for encoding the occurrence of some special condition. We refer to these special values as *optional values* so that they can be distinguished from others non-optional values, the *ordinary values*. A clear example of optional types are pointer types, having `NULL` as the optional value, which encodes the peculiar, though common condition of a pointer pointing to nothing. The purposes of null pointers are essentially the following:

- initialization of a pointer-type resource to a known and comparable value;
- encoding of the fact that a certain condition involving a pointer-type resource has occurred (often, but not always, an error condition);
- allowing the definition of recursive data structures.

As programs grow, the proper handling of optional values and their encoded conditions becomes increasingly difficult. Some programming languages, such as OCaml and Rust, include syntactic constructs allowing the explicit declaration and management of optional types supported by automatic checks performed statically and/or dynamically. As far as C-rusted is concerned, support for the handling of optional types at compile time is offered via `e_opt(...)` annotations.

```
1  #define e_opt_hown() e_opt(NULL) e_hown()
2
3  void * e_opt_hown() e_uninit()
4  malloc(size_t size);
5
6  void * e_opt_hown()
7  calloc(size_t nmemb, size_t size);
8
9  void
10 free(void * e_opt_hown() e_release() ptr);
```

Fig. 4. Ownership and optionality in C Standard Library functions

Figure 4 illustrates the concepts discussed so far by showing the C-rusted interpretation of some of the C Standard Library functions that deal with ownership of heap-allocated memory resources. The `e_opt_hown()` annotation for the return type of `malloc()` and `calloc()` encodes the following information:

1) `e_opt(NULL)`, denoting the fact that an optional value, i.e., a null pointer, is returned in case of allocation failure;
2) `e_hown()`, denoting the fact that an owning reference, i.e., a valid pointer, to a heap-allocated resource is returned in case of allocation success.

Summarizing, the value returned from `malloc()` and `calloc()` is an *optional owning reference to a heap-allocated resource*.[3] This information is used by the *C-rusted Analyzer* to diagnose all occurrences of:

- a dereferencing operation on an optional reference unless guarded by a suitable optionality check (typically encoded by an `if` statement);

[2] We call a C-rusted program *safe* if the *C-rusted Analyzer* does not issue warnings for it.

[3] Note that the C code in Figure 4 is for illustration purposes only: the *C-rusted Analyzer* automatically interprets the C Standard Library functions *as if* they were declared with suitable C-rusted annotations and does not require touching the header files provided by the underlying C implementation.

- a function call using an optional reference as actual parameter unless the call is guarded by a suitable optionality check or the callee's formal parameter is declared to be an optional reference as well;
- a return of an optional reference from a function unless the return statement is guarded by a suitable optionality check or the function return type is declared to be an optional reference as well;
- an (optional or non-optional) owning reference going out-of-scope.

The diagnostic messages require the programmers to concentrate on the intended program logic and take appropriate actions, such as filtering out optional values and/or adding or amending the annotations. When user-defined functions are involved, diagnostics may be resolved:

- in the function call, by providing suitable actual parameters and destination for the return value;
- in the function declaration, by adjusting the annotations of the formal parameters and return type and making sure the function body is coherent with the declaration.

In all cases, this results in increases program readability thanks to the expressive power of annotations, in particular as far as function interfaces are concerned. And, when all diagnostic messages by the *C-rusted Analyzer* have been addressed, the effort is rewarded by strong safety and security guarantees by construction. For instance, optional types are a complete and effective method for tracking the generation and propagation of null pointers: together with the ownership model, the avoidance of invalid pointer dereferencing and invalid deallocations is guaranteed.

Another important aspect of C-rusted is that, unless otherwise specified, all references must refer to initialized resources, thus avoiding uninitialized memory reads. This is the purpose of the `e_uninit()` annotation on the return type of `malloc()` in Figure 4. As `calloc()` returns zero-initialized memory, no such annotation is used for its return type.

As can be seen in Figure 4, `free()` is the designated function for the release of heap-allocated memory resources: this is expressed by the `e_release()` annotation. In combination with `e_opt_hown()`, this means that all calls to `free()` must be performed by passing either

- a null pointer (upon which the `free()` function will do nothing, as guaranteed by the C Standard), or
- an owning reference to a *releasable* memory resource.[4]

Even though C-rusted supports explicit annotations for exclusive and shared references (via the `e_excl(...)` and `e_shar(...)` annotations, respectively), in many cases such annotations are not needed and relying on "const correctness" is sufficient. Namely, the *C-rusted Analyzer* infers the shared or exclusive nature of a reference from the presence or absence of the `const` qualifier on the referred resource, respectively. This is illustrated in Figure 5, which contains declarations for a node (the basic building block of a singly-linked list) and some

[4]In C-rusted parlance, a resource is "releasable" when it has been properly finalized and/or it does not contain any valuable or sensible information.

```
#include <crusted.h>

typedef int T;

typedef Node_t {
  T elem;
  struct Node_t * e_opt_hown() nextp;
} Node_t;

// Shared reference.
void node_print(const Node_t *nodep);
// Exclusive reference.
bool node_insert_after(Node_t *nodep, T elem);
// Owning reference.
Node_t * e_opt_hown() node_ctor(T elem);
```

Fig. 5. Different kinds of references

manipulation functions. The `node_print()` function, which only needs to read the resource "node" in order to print it, correctly takes as parameter a reference to a `const`-qualified resource: in C-rusted, this is interpreted as an implicit shared reference. Note that the concept of shared reference is stronger than `const`-qualification: while in C the constness only affects the directly-referred node, in C-rusted the "shareability" (and the constness property) is recursively propagated down to the last node of the list. Another crucial aspect is that while in C the constness of an object can be easily bypassed, e.g., using pointer casts, in a safe C-rusted program this is not allowed. Going on with the analysis of Figure 5, the first parameter of `node_insert_after()` function (which modifies the list inserting a new node after the referred one) is an example of implicit exclusive reference, whereas the return type of `node_ctor()`, whose purpose is the acquisition of a new node, is an optional owning reference. As a final note on Figure 5, note the inclusion of `crusted.h`, whose purpose is simply to ensure all annotations are removed during the preprocessing translation phase and do not affect the compilation of the program.

All this has an unmistakable Rust taste, of course, generalized to all kinds of resources (not just memory blocks) and all kinds of references (not just pointers).

C-rusted annotations allow expressing much more:

- The fact that library and user-defined functions may encode different information in the same C object. For instance, the return value of POSIX's `open()` is encoded into an `int`, which is either -1, in case of error, or it is a file descriptor.
- Other instances of *nominal typing* fully under control of the programmer.
- The way in which functions modify the properties of resources.

V. Nominal Typing

Nominal typing is a restriction placed by strong type systems whereby two types are compatible only if they have the same name, independently from their underlying representation. In the C world this concept is already present in the MISRA C *essential type model* [2]: a Boolean is not an integer, even when it is represented by an `int`, as it may be the case in C90 implementations [13], [14]. Similarly, an object of enumerated type is not an integer, despite being represented by an implementation-defined integer type. Generally speaking, nominal typing allows imposing a clear separation between the C data type representation and the semantics of the particular type, preventing unwanted and often dangerous operations on nominal types, such as conversions, arithmetic and bitwise manipulation.

```
1  typedef int e_type() e_val(e_geq(0)) fd_t;
2  typedef fd_t e_own() fd_own_t;
3  typedef fd_own_t e_opt(-1) fd_opt_own_t;
4
5  fd_opt_own_t
6  open(const char *path, int oflag);
7
8  int e_val(e_range(-1, 0))
9  close(fd_own_t fildes);
```

Fig. 6. C-rusted view of `open()` and `close()`

Nominal typing is fully supported by C-rusted's type system: as an example, file descriptors are recognized and treated as nominal types. Figure 6 shows how some functions involving file descriptors are interpreted within C-rusted: in addition to the optionality and ownership information conforming to the POSIX specification, the `e_type()` annotation is used to specify that `fd_t` and all the types derived from it are nominal types. Resources of type `fd_t` are file descriptors and, even if at a C level they are represented using integers, they have nothing to do with integers. In particular, they cannot be mixed converting the one to the other, and operations that are permitted on integers are not permitted on file descriptors. Moreover, the `e_val(...)` annotation specifies that:

1) file descriptors are represented by non-negative integers;
2) in case an error occur, `open()` returns the integer -1 (which is not a file descriptor);
3) `close()` returns an integer in the interval $[-1, 0]$.

Note that C-rusted also supports annotations to express the file access mode according to the value of the actual parameter `oflag`, which must be given as a compile-time constant. This allows *C-rusted Analyzer* checking the validity of operations involving a file descriptor (such as `read()` and `write()`). In order to keep the example as short as possible, such annotations have been omitted.

When the power of nominal typing is put into the hands of programmers, a number of applications emerge that have the potential of preventing many programming errors. In Figure 7, four different nominal types related to temperature scales are defined: Celsius, Kelvin, ΔC and ΔK, all of them having `double` as underlying type. This shows how nominal typing can prevent accidentally mixing different temperature scales, independently of the underlying C data types. Therefore, value of nominal types have been constrained within the proper ranges and the admitted operations upon them have been made explicit through `e_bop(...)` (for binary operations) and `e_uop(...)` (for unary operations), so that only meaningful operations are allowed. For instance, while `celsius_t - celsius_t` is admissible as is `kelvin_t - kelvin_t`, and these give `dltcelsius_t` and `dltkelvin_t`, respectively, `celsius_t / celsius_t` must be flagged because it has no physical meaning due to the fact that Celsius is not an absolute scale, whereas `kelvin_t / kelvin_t` makes perfect sense.

VI. Resource Management

In this section we discuss C-rusted ability to capture the way in which functions modify the "properties" of resources: this includes user-defined properties and all sorts of resources.

A. Initialization and Finalization

As already mentioned, in C-rusted all references must refer to initialized resources unless specified otherwise. This means that functions that deal with uninitialized resources must be properly annotated using either `e_init(...)` or `e_uninit(...)`. The former is used to annotate function parameters that are references whose purpose is the initialization of the referred resource: upon entry to the function the resource may be uninitialized, whereas upon exit the resource will be definitely initialized (inside the function body, the *C-rusted Analyzer* will flag all operations on the referred resource that precede initialization). The `e_uninit(...)` annotation is used to annotate possibly uninitialized resources or references to possibly uninitialized resource, as in the case of `malloc()`'s return type.

An application of such concepts is presented in Figure 8, where `e_init()` appears in the annotation of the `channel_ctor()` function parameter. While the first call to `channel_send()` is flagged by the *C-rusted Analyzer* as use of an uninitialized resource, the second call to `channel_send()` is perfectly legal as it takes place after a call to `channel_ctor()`. Figure 8 also shows the use of the `e_fini()` annotation. In line 3, it means that resources of type `channel_t`, once initialized, need to be finalized before being released: failure to do so, as it happens in line 14, triggers a *C-rusted Analyzer* message. In line 7, `e_fini()` identifies the function in charge of the resource finalization.

Note that, for some resources, finalization is crucial. As an example, for resources storing confidential information it is recommended to completely overwrite the used memory locations in order to ensure such information stays in memory for the shortest possible time (a release of the resource memory alone does not achieve that). Of course, in C-rusted a resource

```
#include <crusted.h>

typedef double e_type() e_val(e_geq(-273.15)) celsius_t;    // Celsius.
typedef double e_type() e_val(e_geq(0))       kelvin_t;     // Kelvin.
typedef double e_type()                       dltcelsius_t; // Delta Celsius.
typedef double e_type()                       dltkelvin_t;  // Delta Kelvin.

e_bop(dltcelsius_t, celsius_t, -, celsius_t); // ΔC = C − C.
e_bop(dltkelvin_t, kelvin_t, -, kelvin_t);    // ΔK = K − K.
e_bop(double, kelvin_t, /, kelvin_t);

void bar(celsius_t c1, celsius_t c2, kelvin_t k1, kelvin_t k2) {
  dltcelsius_t dltc = c1 - c2;
  double c_ratio = c1 / c2; // Operation not allowed.
  double k_ratio = k1 / k2;
  // ...
}
```

Fig. 7. Nominal types

```
#include <crusted.h>

typedef struct { /* ... */ } e_fini() Channel_t;

void channel_ctor(Channel_t * e_init() chanp);
bool channel_send(Channel_t *chanp, const char *msg);
void channel_dtor(Channel_t * e_fini() chanp);

int baz(void) {
  Channel_t c;
  channel_send(&c, "..."); // Use of uninitialized resource.
  channel_ctor(&c);
  if (!channel_send(&c, "Message"))
    return -1;             // Missing finalization.
  channel_dtor(&c);
  return 0;
}
```

Fig. 8. Initialization and finalization of resources

that has been finalized is not readable anymore: it can only be re-initialized or released.

B. Custom Properties

Through annotations, C-rusted allows the programmer to also express non-trivial data properties that are bound to the program logic: for example, the *C-rusted Analyzer* is capable of ensuring that a set of user-defined operations are performed in the correct ordering. This is done through the `e_in(...)` and `e_out(...)` annotations expressing preconditions and postconditions, respectively. An example of this facility is presented in Figure 9. There, an opaque type `struct Mixer_t` (the kitchen tool) is declared along with functions that operate on a resource of type `Mixer_t`. Each function declaration specifies the preconditions that must hold on the referred mixer to safely carry out the computation and the postconditions that must hold on the mixer when the function returns to the caller. In function `qux()` the contract specified by the annotation is violated on two accounts. In line 11 there is a violation of `mixer_on()` preconditions because the door may be open: `e_in(blade=off)` states that it will take in input a reference to a mixer having the blades turned off, no preconditions about the state of the door are present. Furthermore, function `qux()`, in line 16, is also violating the postconditions of its own signature: `e_out(door=?)` states that, upon return, the door of the passed mixer may be open or closed; nothing is said about the state of the blade upon return, which implies the blade return state must be equal to the entry state, i.e., the

```c
#include <crusted.h>

typedef struct Mixer Mixer_t;

void mixer_open(Mixer_t * e_in(blade=off) e_out(door=opened) mxp);
void mixer_close(Mixer_t * e_out(door=closed) mxp);
void mixer_on(Mixer_t * e_in(door=closed) e_out(blade=on) mxp);
void mixer_off(Mixer_t * e_out(blade=off) mxp);

void qux(Mixer_t * e_in(blade=off) e_out(door=?) mxp) {
  mixer_on(mxp);    // Door may be open!
  mixer_close(mxp);
                    // Door closed.
  mixer_on(mxp);
                    // Blade on.
  return;           // Blade still on!
}
```

Fig. 9. Preconditions and postconditions for the safe use of a mixer

blade must be turned off upon return. The problem is that this is not happening: the programmer has probably forgotten to call `mixer_off()` before returning from the function. These mistakes are reported in the form of compile-time warnings by the *C-rusted Analyzer*.

VII. Safe and Unsafe Boundaries

C-rusted allows enforcing information hiding and a sharp separation between interface and implementation by means of a flexible access restriction system based on the `e_unsafe(...)` annotation, which identifies data types, functions and operations that are "unsafe" on their own or are considered unsafe because they encapsulate and/or use other unsafe entities: in this context "unsafe" means "requiring special care and knowledge in order to ensure safety."

An example where this is applied concerns the pointers to the `FILE` objects used to control the standard I/O streams. The application programmer obtains such pointers by calling the `fopen()` standard function, but these ought to be treated as if they were not pointer at all: just atomic, unique identifiers with a `NULL` special value. If they were implemented as opaque pointers some of the potential issues (e.g., copies of a `FILE` object may not give the same behavior as the original) would be prevented, but there is no such a guarantee. In fact, MISRA C has a mandatory rule that bans dereferencing pointers to `FILE` [2, Rule 22.5]. Figure 10 shows how the `fopen()` and `fclose()` functions are seen by C-rusted: the `e_unsafe("FILE")` annotation ensures that, by default, all accesses to `FILE` objects are flagged by the *C-rusted Analyzer*.[5] Note that the string literal argument in `e_unsafe("FILE")` is arbitrary, which allows an unlimited number of "unsafety kinds."

[5]As in the case of `open()`, for brevity the annotation of `fopen()` shown in Figure 10 omits the specification of the file access mode encoded in the `mode` actual parameter, which must be given as a string literal.

```c
e_decl_props(FILE, e_unsafe("FILE"));
typedef FILE * fp_t;
typedef fp_t e_own() fp_own_t;
typedef fp_own_t e_opt(NULL) fp_opt_own_t;

fp_opt_own_t
fopen(const char * restrict filename,
      const char * restrict mode);

int e_val(e_eq(0) || e_eq(EOF))
fclose(fp_own_t fp);
```

Fig. 10. C-rusted view of `fopen()` and `fclose()`

For the implementation side, C-rusted provides two annotations: `e_unchecked(...)` and `e_checked(...)`. `e_unchecked(...)` marks a statement as not expected to conform to the C-rusted safety and security requirements: every function containing unchecked statements must thus be annotated as unsafe. `e_checked(...)` also marks a statement as not expected to conform to the C-rusted syntax and semantics, but its use is guaranteed to be safe by the programmer under every aspect of C-rusted needed warranties. An example is presented in Figure 11 where, in order to correctly implement the `fclose()` function, all the accesses to a `FILE` object are encapsulated within the proper safety annotation as it happens in Line 11. As a result, under the responsibility of implementers, function `fclose()` will be considered as safe by the *C-rusted Analyzer*.

This model is powerful, flexible and can be used to cover similar types, such as type `sem_t` of the POSIX Library, and user-defined entities, such as the type `Channel_t` of Figure 8. For the latter, this will ensure:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include "local.h"
#include <crusted.h>

int e_val(e_eq(0) || e_eq(EOF))
fclose(fp_own_t fp) {
  // ...

  e_checked("FILE") {
    if (fp->flags == 0U) {
      errno = EBADF;
      return EOF;
    }
  }

  // ...
}
```

Fig. 11. Fragment of `fclose()` implementation with C-rusted annotations

1) the use of the communication channel only through the safe interfaces;
2) the access to the (delicate) implementation details only by the channel implementers.

Note how this approach leads to the correct propagation and, at the same time, the correct encapsulation of (possibly) unsafe operations within the proper safety checks.

VIII. Discussion

C-rusted is a pragmatic and cost-effective solution to up the game of C programming to unprecedented integrity guarantees without giving up anything that the C ecosystem offers today. That is, keep using C, exactly as before, using the same compilers and the same tools, the same personnel… but *incrementally* adding to the program the information required to demonstrate correctness, using a system of annotations that is not based on mathematical logic (or other complex languages) and can be taught to programmers in a week of training.

This technique is not new: it is called *gradual typing*, and consists in the addition of information that does not alter the behavior of the code, yet it is instrumental in the verification of its correctness. Gradual typing has been applied with spectacular success in the past: Typescript has been created 10 years ago, and in the last 6 years its diffusion in the community of JavaScript developers has increased from 21% to 69%. And it will continue to increase: simply put, there is no reason for writing more code in the significantly less secure and verifiable JavaScript language [15].

For C, a similar approach is the one of *Checked C* [16]. There, gradual typing is used to extend C with static and dynamic checking aimed at detecting or preventing buffer overflows and out-of-bounds memory accesses. Checked C supports annotations for pointers and array bounds and the use of static analysis to validate existing annotations and to infer new ones. Note, though, that Checked C is a different language than C: while the compilation of Checked C code requires a special compiler, compilation of C-rusted code is done with any C compiler.

Figure 12 places C-rusted in its context, between C and Rust, and summarizes the main elements for a comparison. Some of these points deserve further explanation.

First, C-rusted is not a new programming language, like Rust and Zig: C-rusted code is standard ISO C code just used in a peculiar way and in association with suitable static analysis techniques.[6] As such, C-rusted benefits from the huge investment the industry has made into C in terms of compilers, tools, developers, coding standards and code bases.[7] For instance, C-rusted is 100% compatible with MISRA C: a C program that is MISRA compliant can be *rusted* without negatively impacting MISRA compliance. Furthermore, an annotated C-rusted program validated by the *C-rusted Analyzer* has strong guarantees of compliance with respect to guidelines, such as those concerning the disciplined use of resources, error handling and possibly tainted inputs, for which compliance is much harder to achieve and argument in other ways.

Functional safety standards such as ISO 26262 [20] prescribe the use of safe subsets of standardized programming languages used with qualifiable translation toolchains (see, e.g., [21] and [22]). Insofar a C-rusted program is a standard ISO C program where the presence of annotation does not invalidate MISRA compliance, C-rusted fits the bill as C does and more, due to the strong guarantees provided by annotations. Contrast this with Rust and Zig: they are not standardized and, as a matter of fact, they frequently change in a way that does not follow a rigorous process. This is the main reason why qualifying a Rust or Zig compilation toolchain according to major functional safety standards is, in the authors' opinion, impossible today. In contrast, any qualified C compiler is, as is, a qualified C-rusted compiler.

C-rusted has been conceived for incremental adoption: C programs can be (partly) annotated so as to express: ownership, exclusivity and shareability of language, system and user-defined resources, as well as properties of resources and the way they evolve during program execution. The annotated C-rusted program parts can be validated by static analysis: if the *C-rusted Analyzer* flags no error, then the annotations are provably coherent among themselves and with respect to annotated code, in which case said annotated parts are provably exempt from a large class of logic, security, and run-time errors. C-rusted can thus prevent many resource management errors: missing allocation, missing initialization, missing deallocation (resource leak), use after deallocation,

[6] C-rusted is compatible with any version of the ISO C Standard and can be used with any C toolchain.

[7] We note on passing that, in the authors' opinion, C-to-Rust transpilation [17], [18], [19] is not a real solution: transpiling well-written C code to unreadable and unmaintainable Rust code could possibly solve only a small fraction of the problems at the cost of creating several new problems. This, however, goes beyond the scope of this paper.

	C	C-rusted	Rust
Standardized	Yes: ISO	Yes: it *is* ISO C	No: moving target
Certifiable translators exist	Yes	Yes: it *is* ISO C	No
Portability	Absolute	Absolute	Limited
Tool availability	Very large	Very large	Scarce
Developers' availability	Large	Large	Scarce
Coding standards for safety and security	Yes	Yes	No
Can reuse C legacy code		Yes	Only in some cases
Strong guarantees on memory resources for annotated programs		Yes	Yes
Strong guarantees on user-defined resources for annotated programs		Yes	Yes
Compatibility with unannotated code		Yes	Yes
Incremental adoption		Yes	No
Cost of retraining C programmers for unannotated code		Zero	Significant
Cost of retraining C programmers for annotated code		Moderate	Significant

Fig. 12. Advantages and disadvantages of C-rusted (along with its C inheritance) and Rust

multiple deallocation, race conditions due to sharing. And this on all sorts of resources:

Language-defined resources: e.g., memory blocks, stream-controlling objects, mutexes.

System resources: e.g., open file descriptions, streams, sockets.

User-defined resources: all sorts of transactions, anything that requires allocation, deallocation and disciplined exclusive and/or shareable use.

Thanks to nominal typing/subtyping and to the tracking of properties, C-rusted can also prevent other errors not related to the management of resources, such as the missing detection of erroneous or anomalous conditions, the use of possibly tainted input data, and the unwanted disclosure of confidential information.

C-rusted has been conceived for incremental adoption: new code that is critical can be created with annotations from the outset, and this will speed up development because the *C-rusted Analyzer* will immediately provide warnings about a large class of mistakes. Legacy code can be annotated later, if there is value in doing so, or even left unannotated forever: touching proven-in-use code with an honorable operational history makes no sense. Note that annotations are not intrusive: they can be embedded into typedefs and, for a large part, they are confined to function prototypes and declarations of structs containing references.

IX. Implementation

The implementation of the *C-rusted Analyzer* is based on the *ECLAIR Software Verification Platform*.

The static analysis component is formalized in terms of *abstract interpretation* [23]. The analysis is rigorously *intraprocedural*, i.e., it is done one function at a time, using only the information available for that function in the translation unit defining it, which includes the annotations possibly provided in function declarations.

The analysis domains include a very precise flow-sensitive and field-sensitive points-to analysis. Other analyses involve variable liveness and the tracking of numeric information through value range analysis based on constraint propagation over multi-intervals. In addition, there are several finite domains specifically conceived for C-rusted, which track the state of resources and references as well as the evolution of dynamic semantic properties. Scalability is ensured by intraprocedurality.

All the annotations of C-rusted are realized via macro invocations: the corresponding macros all expand to the empty token sequence (with the exception of global annotations, which expand to something that obeys ISO C syntax and is ignored by the compiler) so that, as far as the compiler is concerned, after translation phase 4 [1, Section 5.1.1.2] it is as if they never existed. Of course, the *C-rusted Analyzer* uses all the information provided by the annotations before letting the preprocessor making them vanish.

References

[1] ISO/IEC, *ISO/IEC 9899:2018: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2018.

[2] MISRA, *MISRA C:2023 — Guidelines for the use of the C language critical systems*. Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Apr. 2023, third edition, Second revision.

[3] P. Baudin, F. Bobot, F. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, "The dogged pursuit of bug-free C programs: The Frama-C software analysis platform," *Communications of the ACM*, vol. 64, no. 8, pp. 56–68, 2021.

[4] D. M. Ritchie, *The Development of the C Programming Language*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 671–698. [Online]. Available: https://doi.org/10.1145/234286.1057834

[5] J. Salter, "Linus Torvalds weighs in on Rust language in the Linux kernel," *Ars Technica*, Mar. 2021. [Online]. Available: https://arstechnica.com/gadgets/2021/03/linus-torvalds-weighs-in-on-rust-language-in-the-linux-kernel/

[6] B. Cantrill, "Is it time to rewrite the operating system in Rust?" *InfoQ*, Jan. 2019. [Online]. Available: https://www.infoq.com/presentations/os-rust/

[7] J. Wallen, "Let the Linux kernel Rust," *TechRepublic*, Jul. 2021. [Online]. Available: https://www.techrepublic.com/article/let-the-linux-kernel-rust/

[8] S. J. Vaughan-Nichols, "Linus Torvalds on where Rust will fit into Linux," *ZDNet*, Mar. 2021. [Online]. Available: https://www.zdnet.com/article/linus-torvalds-on-where-rust-will-fit-into-linux/

[9] ——, "Where Rust fits into Linux," *The Register*, Nov. 2021. [Online]. Available: https://www.theregister.com/2021/11/10/where_rust_fits_into_linux/
[10] L. Tung, "Google backs effort to bring Rust to the Linux kernel," *ZDNet*, Apr. 2021. [Online]. Available: https://www.zdnet.com/article/google-backs-effort-to-bring-rust-to-the-linux-kernel/
[11] M. Melanson, "Rust in the Linux kernel: 'good enough'," *The New Stack*, Dec. 2021. [Online]. Available: https://thenewstack.io/rust-in-the-linux-kernel-good-enough/
[12] N. Elhage, "Supporting Linux kernel development in Rust," *LWN.net*, Aug. 2020. [Online]. Available: https://lwn.net/Articles/829858/
[13] ISO/IEC, *ISO/IEC 9899:1990: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1990.
[14] ——, *ISO/IEC 9899:1990/AMD 1:1995: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1995.
[15] P. Krill, "TypeScript usage growing by leaps and bounds — report," *InfoWorld*, Feb. 2022. [Online]. Available: https://www.infoworld.com/article/3650513/typescript-usage-growing-by-leaps-and-bounds-report.html
[16] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, "C to Checked C by 3c," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, Apr. 2022. [Online]. Available: https://doi.org/10.1145/3527322
[17] N. Shetty, N. Saldanha, and M. N. Thippeswamy, "CRUST: A C/C++ to Rust transpiler using a "nano-parser methodology" to avoid C/C++ safety issues in legacy code," *Emerging Research in Computing, Information, Communication and Applications*, 2019.
[18] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.
[19] M. Ling, Y. Yu, H. Wu, Y. Wang, J. Cordy, and A. Hassan, "In Rust we trust — a transpiler from unsafe C to safer Rust," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 354–355.
[20] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.
[21] ——, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 8: Supporting processes*. Geneva, Switzerland: ISO, Dec. 2018.
[22] RTCA, SC-205, *DO-330: Software Tool Qualification Considerations*. RTCA, Dec. 2011.
[23] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*. Los Angeles, CA, USA: ACM Press, 1977, pp. 238–252.