embeddedworld
Exhibition&Conference

C-rusted: A Formally Verifiable Flavor of C for the Development of Safe and Secure Systems

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

Abramo Bagnara, Nicola Vetrini
BUGSENG, Italy
Email: name.surname@bugseng.com

Federico Serafini
BUGSENG, Italy
Email: name.surname@bugseng.com
Ca' Foscari University of Venice, Italy
Email: name.surname@unive.it

***Abstract*—C-rusted is an innovative technology whereby C programs can be (partially) annotated so as to express: ownership, exclusivity and shareability of language, system and user-defined resources; properties of objects and the way they evolve during program execution; optional types, nominal types and subtypes compatible with any C data type. The annotated C programs, being fully compatible with all versions of ISO C, can be translated with unmodified versions of any C compiler. The crucial point is that such annotations express program properties that can be formally verified through static analysis: if the static analyzer flags no error, then the annotations are provably coherent among themselves and with the annotated C code, in which case the analyzed program portions are provably exempt from a large class of logic, security, and run-time errors. The annotation system has been designed not to be intrusive: in most of the situations, also the lack of annotations has a precise meaning that is used by the static analyzer to formally verify program properties. 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 formally verify correctness, using a system of annotations that is not based on complex formalisms (such as mathematical logic) and can be taught to programmers in a week.**

I. Introduction

A. The Spirit of C

The C programming language was designed to be simple, minimal and fast: no whistles or frills, just pragmatic necessities for the development of the UNIX operating system for a PDP-11 computer. The spirit of C is clearly spelled out in the C charter [1]:

(a) Trust the programmer.
(b) Don't prevent the programmer from doing what needs to be done.
(c) Keep the language small and simple.
(d) Provide only one way to do an operation.
(e) Make it fast, even if it is not guaranteed to be portable.
(f) Make support for safety and security demonstrable.

From the moment of its inception in the 1970s, C gradually gained popularity until it has become a crucial foundation of all current applications of information technology:

- The way C is defined simplifies the task of producing optimizing compilers: this is the reason why C compilers exist for almost any processor.
- Compiled code is very efficient and without hidden costs: this makes C suitable to applications where high performance is crucial.
- The syntax allows writing compact code thanks to the many built-in operators and the limited verbosity of its constructs.
- C is defined by an ISO standard [2].
- C, possibly with extensions, allows easy access to the hardware.
- C provides the basis upon which the implementations of many other programming languages and their run-time environments are built.
- C has a long history of use, including in critical systems.
- C is widely supported by all sorts of tools.

As a direct consequence, there are strong economic reasons behind the use of C and it has no equals as long 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.

B. The Other Side of the Coin

The efficiency and simplicity of C obviously come at a cost. 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 instruction sets differ from one architecture to the other, the behavior of C programs is not fully defined;
- the fact that there is nothing happening under the hood also means that no run-time checks are performed to ensure safety and/or security.

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:

- memory references are raw pointers carrying no information about the associated memory block or its intended use;
- 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.

As an example, consider the program in Figure 1 which the GNU C compiler compiles without any warning even at a very high warning level: the task of identifying all the problems is left as an exercise for the reader.

```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 = malloc(bufsize);
  ++fd;
  ssize_t bytes = read(fd, buf, bufsize);
  buf[bytes] = '\0';
  process(buf);
  return 0;
}
```

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

Nowadays, increasing attention is paid to safety and security of software systems and the C programming language is often criticized for the ease with which programming mistakes are committed, especially those related to memory management that can lead to safety and security vulnerabilities. Memory safety is currently a hot topic to the point that the *White House Office of the National Cyber Director* released a report to promote a shift toward memory-safe programming languages [3].

Throwing away the C ecosystem (which is worth zillions) is completely impractical. The main issues are as follows:

- 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 embedded systems, there are currently no alternatives to C.
- Tools: while all sorts of tools are available for C, the same thing cannot be said for other languages.

C. From C to C-rusted

In this paper, we show how concepts that have been proven to work well in other languages, such as "the ownership model" and "borrowing" of Rust, can be ported to C, in order to define *C-rusted*: a formally verifiable flavor of C for the development of safe and secure systems. 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) optional types, nominal types and 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) Differently from the compiler, a static analyzer —which we will call *C-rusted Analyzer* from now on— does 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 the 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 to learn other languages, apart from those that, working on safety-critical components, would have to get familiar with the annotations.
- Existing C compilers can be used without any change, thereby ensuring maximum portability.
- All sorts of tools, in addition to compilers, can also be used without any change.

Before going into detail about C-rusted, a brief demonstration of its capabilities is given in Figure 2, where the

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

// The actual argument shall be a valid reference (not null)
// to a C string in the heap of which process() will take ownership:
// the caller shall own the resource, otherwise it would be unable to pass it on.
extern void process(char * e_hown() string);

// The first actual argument shall be a valid shared reference (not null) to a C string.
// The second actual argument shall be a constant or an initialized unsigned variable.
int foo(const char *fname, size_t 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 a file.
  if (fd == -1)
    return 1;
  // `fd' is definitely an owning reference to a file.

  char *buf = malloc(bufsize);
  // `buf' is either NULL or an owning reference to an uninitialized heap-allocated array.
  if (buf == NULL || bufsize == 0U) {
    (void) close(fd);
    // Ownership of the file moved from the actual argument
    // 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 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 argument
    // 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 argument to formal parameter, as in line 22.
    return 1;
  }
  // `bytes' is definitely the number of bytes read into `buf'.
  buf[bytes] = '\0';
  // `buf' is definitely an owning reference to a C string in the heap.

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

  (void) close(fd); // Ownership moved from actual argument to formal parameter, as in line 22.
  return 0;
}
```

Fig. 2. A C-rusted program for which the *C-rusted Analyzer* gives no warnings

logical argument followed by the *C-rusted Analyzer* to validate the program and consider it *safe*[1] is put into words through comments in the code:[2] readers familiar with both C and Rust will find several similarities, while the others can find an explanation of all the concepts in Sections II and III, but please note that:

- While `fd` is declared having type `int`, the value of `fd` has properties that change throughout the function body; similarly for `buf` and `bytes` and `bufsize`. In other words, the C-rusted type system is able to track how properties of resources change depending on the program point.
- A large part of the guarantees are obtained without any user 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).
- Annotations are not heavy and do not clutter the code: they are mostly limited to function declarations and type qualifiers such as `e_hown()` can also be embedded in typedefs; a proper choice of typedef names also helps readability and understandability.

II. Problems and Solutions

In the following sections some of the most common C memory and resource-management issues will be briefly discussed: for each of them, the solution implemented by C-rusted will be illustrated with the help of simple yet realistic examples.

A. Null Pointer Dereferencing

The reasons behind the existence of null pointers in C are the following:

- allowing the default initialization of pointer-typed variables to a known value;
- allowing the encoding, through a special value, of the fact that a certain condition has occurred (which is often, but not always, an error condition);
- allowing the definition of recursive data structures by means of pointers that can eventually point to nothing.

In C, dereferencing a null pointer is undefined behavior. This means that the C standard places no limit to what a program dereferencing a null pointer can do. A segmentation fault raised by the hardware is the best outcome that one can expect from executing a program that dereferences a null pointer, but there is no guarantee that this is what will happen: the program could silently corrupt the memory, continue running, and exhibit erratic behavior of any kind.

The fact that today, as has been the case since the language went into existence, C programs are exposed to the problem of null-pointer dereferencing is mainly due to the following factors:

1) the C programming language is not meant to get in the way of the programmer: forcing the implementation to perform null-pointer checks goes against the spirit of C: trust the programmer to know whether these checks are required and compatible with the efficiency requirements of the program at hand;
2) for the sake of efficiency, the tendency of programmers is to only code the null-pointer checks that they deem strictly necessary, but they frequently get it wrong and things get worse as the program grows.[3]

In C-rusted a pointer can be null only if it is annotated as such through the `e_opt(NULL)` annotation: such pointer is said to be an *optional pointer*. The concept of optionality, however, covers more than just pointers: an *optional type* is a type annotated with `e_opt()`, so as to identify 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 other non-optional values, the *ordinary values*. For pointers, the value `NULL` is a clear example of optional value.

Identifying optional types through annotations allows the *C-rusted Analyzer* to support the programmer in the proper handling and propagation of the corresponding optional values (passed as arguments to the annotation) through the following constraints, which are enforced at compile time:

- a dereferencing operation of an optional pointer shall be permitted only if it is guarded by an explicit test (typically in the guard of an `if` statement or an iteration statement) which filters the optional null value out;
- the assignment of an expression having an optional type shall be permitted only if the destination has an optional type as well, unless the assignment is guarded as above; this applies also to passing arguments to functions, returning values from functions, and initializations.[4]

As an example, in Figure 3 the `find()` function is defined as returning a pointer to the first occurrence of the element `elem` in `vec`, or a null pointer in case the element is not found. For that program, the *C-rusted Analyzer* will report two warnings to highlight the wrong management of optional types: the first one at line 8, where the function returns `NULL` despite the return type not being an optional pointer; and the other one at line 13, where the optional pointer `vec` is used as an actual argument of function `find()` where, and this is the problem, the corresponding formal parameter is not optional. [5] The diagnostic messages require the programmers to concentrate on the intended program logic and take appropriate

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

[2] In this document, "shall" and "shall not" need to be interpreted as requirements: a program that does not meet all the requirements is considered to be unsafe as it will generate warnings when fed to the static analyzer.

[3] Note that such observations apply also for other problems such as out-of-bounds accesses.

[4] In this, we follow MISRA C that, when using the word 'assignment' also covers argument passing (which assigns the value of the actual argument to the formal parameter), returning an expression from a function (which assigns the returned value to a corresponding slot in the caller's activation record), and using an expression to initialize all or part of an object [4, Glossary].

[5] If no argument is given to `e_opt( )`, the default semantics is to consider zero as the optional value. This decision was made taking into account the many uses of such annotation on pointer types: annotating a pointer type with `e_opt( )` is thus a shortcut for `e_opt( )NULL`.

```
#include <crusted.h>

T *find(T *vec, size_t n, T elem) {
  for (size_t i = 0; i < n; ++i)
    if (vec[i] == elem)
      return &vec[i];

  return NULL;
}

void foo(T * e_opt() vec, size_t n, T elem) {

  T *ptr = find(vec, n, elem);

  // ....
}
```

Fig. 3. Bad handling of null pointers

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

In all cases, this results in increased 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. Optional types are a complete and effective method for tracking the generation and propagation of null pointers.

A correct definition and use of function `find()` is presented in Figure 4, which contains the required optionality annotations and checks. Note that:

1) In the body of `find()`, optionality checks for formal parameter `vec` are not required, being it a non-optional pointer (any call to `find()` passing a possibly null pointer will raise a warning).
2) In the body of `find()`, at line 15 there is no need to explicitly mark the type of `ptr` as optional because it is a *naked type*, i.e., a type without any C-rusted annotations. In fact, when an operand with naked type appears as the left operand of an assignment operator, the *C-rusted Analyzer* implicitly "transfers" to it the annotations found in the type of the right operand. This form of type inference saves typing without endangering safety.
3) For the program in Figure 4 no warnings involving optional types are reported and, as a consequence, guarantees about the absence of null-pointer dereferencing are given at compile time by construction.[6]

```
#include <crusted.h>

T * e_opt() find(T *vec, size_t n, T elem) {
  for (size_t i = 0; i < n; ++i)
    if (vec[i] == elem)
      return &vec[i];

  return NULL;
}

void foo(T * e_opt() vec, size_t n, T elem) {

  if (vec == NULL) return;

  T *ptr = find(vec, n, elem);

  if (ptr == NULL) return;

  // Do something with `ptr'.
}
```

Fig. 4. Good handling of null pointers

B. Memory Leaks, Double Free, Invalid Free, Use After Free

The explicit release of dynamically-allocated memory is the source of many memory management errors:

- memory leaks (deallocation is too late or never happens);
- double free (deallocation of memory that was already deallocated);
- invalid free (attempted deallocation memory that was not allocated);
- use after free (deallocation is too early).

Of those, memory leaks are the only issue that does not affect safety.[7] For all the others, the program behavior is undefined. This is one of the reasons why coding standards for the development of safety-critical systems recommend against or strictly regulate dynamic memory allocation. In MISRA C:2023, a required directive prevents the use of any form of dynamic memory allocation (however implemented), while a required rule explicitly targets the memory allocation and deallocation functions of `<stdlib.h>`, which can only be used if accompanied by a suitable safety argument [4, Dir 4.12, Rule 21.3].

The Rust programming language is best known for its new approach to memory management: the ownership model [5]. C-rusted ports such model to the C language through its annotation language, which allows expressing constraints on

[6]This does not imply that the program does not trigger other warnings: in fact it will trigger warnings related to missing bound annotations (see Section II-E).

[7]It is a resource management issue that may eventually result into a memory allocation failure, an issue that has to be handled anyway, typically by suitable null pointer checks as in the previous section.

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; manipulated by operations that have to be performed in some predefined order; and destroyed or deallocated or unreserved. C-rusted supports different kinds of resources: memory resources and abstract resources (i.e., 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 examples of references.

The C-rusted solution for memory leaks, double free, invalid free, and use after free is based on *owning references*. An *owning reference* is a special kind of reference annotated with `e_hown()` or `e_own()`. Every resource subject to dynamic release (as opposed to automatic release, as in the case of stack variables going out of scope) shall, at all times, be associated to one and only one owning reference and vice versa: the owner is the only one responsible for the release of the owned resource.

The association between the owner and the owned resource is therefore regulated by the following constraints:

- The owned resource shall not outlive its owner. In particular, before the owner goes out of scope its association with the owned resource shall be terminated, releasing the owned resource by means of a call to a designated release function using the owner as actual argument.
- Through the program evolution, the owner of a resource might change, due to a mechanism called *ownership move*: the ownership of a resource is moved from an owner to a new owner due to a copy of the owning reference. Assignments cause an ownership move if both the source and the destination types are owning references.
- An owner that loses ownership shall not be used anymore.

```
1 #define e_opt_hown() e_opt() e_hown()
2
3 void * e_opt_hown() e_size(nmemb*size)
4 calloc(size_t nmemb, size_t size);
5
6 void free(void * e_opt_hown() e_release() ptr);
```

Fig. 5. Ownership and optionality for some memory allocation and deallocation functions of `<stdlib.h>`

The ownership model is implicit in the memory allocation and deallocation functions of `<stdlib.h>`. Figure 5 shows the C-rusted interpretation of some of these functions, namely:

- the pointer returned by `calloc()` is interpreted as if it was annotated with `e_opt_hown()`, denoting an optional owning reference to a heap-allocated resource (the meaning of annotation `e_size()` will be discussed in Section II-E);
- the function responsible for the deallocation of heap-allocated resources is the `free()` function; it is interpreted as taking an optional[8] owning reference to a heap-allocated resource as parameter, which commits the function to releasing the resource, if any.

```
1  #include <crusted.h>
2
3  void do_things(T * e_hown() p);
4
5  void foo(size_t n) {
6
7    T *ptr = malloc(n * sizeof(T));
8
9    if (ptr == NULL)
10     return;
11
12   T *q = ptr;
13
14   *ptr = 42;
15
16   *q = 7;
17   do_things(q);
18
19   free(q);
20
21   q = NULL;
22 }
```

Fig. 6. Ownership move

Some instances of ownership move are presented in Figure 6: the ownership is moved from the returned value of `malloc()` to `ptr` at line 7; then another move happens at line 12 from `ptr` to `q`; the last move is at line 17 from actual argument `q` to the formal parameter of the `do_things()` function, which will take care of releasing the heap-allocated resource. This program is not a safe C-rusted program: the use of `ptr` at line 14 is flagged by the *C-rusted Analyzer* because it is a use of a former owner that lost ownership; similarly, the use of `q` at line 19 is flagged, as it would result in a double free error.

Aside from "heap ownership", C-rusted offers the `e_own()` annotation (note the missing 'h') to denote a different kind of ownership, such as ownership of abstract resources and ownership of dynamic memory allocated through different, user-defined allocators. As an example consider the `fopen()` and `fclose()` functions of `<stdio.h>`: the former is interpreted in C-rusted as if returning an optional owning reference to a `FILE` object, while the latter is considered the release function for such kind of ownership. A similar use of ownership concerns the POSIX library functions `open()` and `close()`. Such file-related functions will be discussed in more detail in the sequel.

[8]Calling `free()` with a null pointer is well defined in C and has no effect.

The key point is that the ownership model gives by construction guarantees about the absence of memory leaks, double free, invalid free and use after free. A program for which the *C-rusted Analyzer* does not give any ownership-related warnings is a program that follows the ownership model.

C. Aliasing

In general, *aliasing* occurs when two or more expressions, the *aliases*, denote the same entity. Dealing with aliasing is tricky and can easily lead to programming errors.

Consider for example the program in Figure 7: at first sight, function `add3()` seems to correctly make use of function `add2()` to compute the sum of three elements. So, the `do_math()` function should print the sum of `x`, `y` and `z`, which is 6, but in fact it prints 8! Clearly, function `add3()` has not been designed taking into account the possibility of aliasing among its formal parameters.

```
1  #include <crusted.h>
2
3  void add2(int *r, int *a, int *b) {
4    *r = *a + *b;
5  }
6
7  void add3(int *r, int *a, int *b, int *c) {
8    add2(r, a, b);
9    add2(r, r, c);
10 }
11
12 void do_math(void) {
13   int x = 2, y = 2, z = 2;
14   add3(&z, &x, &y, &z);
15   printf("%d\n", z);
16 }
```

Fig. 7. Aliasing is tricky

Aliasing makes programs difficult to understand, for humans, and difficult to optimize, for compilers. Moreover, as the aliasing problem is undecidable, aliasing analysis algorithms are necessarily approximated, which, in turn, has implications on the precision of other analysis such as live variables, available expressions (which is used to perform common subexpression elimination) and constant propagation. In summary, limiting the possibility of aliasing in ways that are checkable at compile time is beneficial in a number of ways.

Note that, in C-rusted, the problem of aliasing for resources subject to dynamic release is partially addressed by the ownership model through the mechanism of ownership move. But what about other kinds of resources? And what about referencing a resource without taking the ownership of it?

These concepts are known in Rust under the name of "borrowing" and are implemented in C-rusted distinguishing between different kinds of references: in addition to *owning* references we have *exclusive* and *shared* references, identified by `e_excl()` and `e_shar()` annotations, respectively.

An *exclusive reference*, as the name suggests, grants exclusive access to a resource and, as a consequence, both read and write operations are allowed through the exclusive reference.

Exclusive access to resources through exclusive references is therefore subject to the following constraint: the existence of a *usable* exclusive reference to a resource is incompatible with the existence of any other *usable* reference (of any kind) to the same resource. The notion of *usability* of a reference is rather technical and is beyond the scope of this paper. Suffice it to say that, when multiple references to the same resource exist and at least one of them is *owning* or *exclusive*, *owning* and *exclusive* references are *paused* so as to satisfy the constraint. For example, an *owning* reference is *paused* as soon as another *exclusive* or *shared* reference is created to the same resource: when such new reference ceases to exist, the *owning* reference becomes *usable* again.

```
1  #include <crusted.h>
2
3  void bar(int *q, size_t n);
4
5  void foo(size_t n) {
6    int *ptr = calloc(n, sizeof(int));
7
8    if (ptr == NULL) return;
9
10   int * e_excl() r = ptr;
11
12   free(ptr);
13
14   bar(r, n);
15 }
```

Fig. 8. Exclusive references

Some examples of interaction between owning and exclusive references are presented in Figure 8: at line 10 the exclusive reference `r` is created and `ptr` is paused; at line 12, however, `ptr` is used and this generates a warning: `ptr` will become usable again only after the last use of `r` at line 14.

A *shared reference* to a resource, instead, allows 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, as previously said, during the existence of a shared reference, any exclusive references to the same resource, if any, are not usable.

All constraints related to owning, exclusive and shared references are checked at compile time by the *C-rusted Analyzer* so that, if no warnings are given, problems related to aliasing are excluded. The only cases of aliasing that will survive this screening involve shared references only, which, due to their read-only nature, are harmless.

Even though C-rusted supports the explicit annotations to denote exclusive and shared references, in many cases such annotations are not needed as C-rusted relies on type inference based on the presence/absence of the `restrict` and `const` qualifiers. Namely, in case of pointers, the *C-rusted Analyzer* infers the exclusive or shared nature of a reference by applying, in order, the following steps:

1) if the pointer is restrict-qualified, then it is an exclusive reference;
2) otherwise, if the pointer is not restrict-qualified, then:
   a) it is a shared reference if the pointee is const-qualified;
   b) it is an exclusive reference, otherwise.

```
1  #include <crusted.h>
2
3  typedef struct {
4    T elem;
5    struct Node_t * e_opt_hown() next;
6  } Node_t;
7
8  void node_print_all(const Node_t *nodep);
9  bool node_insert_after(Node_t *nodep, T elem);
10 Node_t * e_opt_hown() node_ctor(T elem);
11 void node_dtor(Node_t * e_opt_hown() nodep);
```

Fig. 9. References in a singly-linked list

This is illustrated in Figure 9, where different references are used to implement nodes of a singly-linked list and some manipulation functions. The `node_print_all()` function, which only needs to read "node" resources in order to print them, correctly takes as argument 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.

Resuming the analysis of Figure 9, the first parameter of the `node_insert_after()` function (which modifies the list by 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 the formal parameter of the destructor `node_dtor()`.

Going back to the tricky aliasing example of Figure 7, note that the program is not considered a safe C-rusted program: at line 14 a warning regarding the first and the last actual arguments of `add3()`'s call is issued because all the actual arguments are implicitly exclusive references, thereby involving the creation of multiple exclusive references to the same resource `z`, which is flagged by the *C-rusted Analyzer*. For the same reason, a warning is reported at line 9 as well.

Summing up: programs that stick to the ownership model and correctly use exclusive and shared references not only are less prone to coding errors related to aliasing: they also have a simpler logic that possibly enables more optimization opportunities for the compiler.

D. Invalid Pointers

The notion of *invalid pointer* includes:

- wild pointers (uninitialized pointers);
- misaligned pointers (pointer storing an address inappropriately aligned for the type of object pointed to);
- dangling pointers (pointer to an object that reached the end of its lifetime).

Using an invalid pointer is undefined behavior.

In a safe C-rusted program, the use of wild pointers is something that cannot happen because all possible usage of uninitialized resources (not only pointers) are flagged. This is possible, also when the code is not available, thanks to `e_uninit()` annotation that is used to explicitly mark memory resources passed to or returned by functions as possibly uninitialized. An example of that is the memory allocated after a successful call to `malloc()` function: `malloc()`'s return type is interpreted by the *C-rusted Analyzer* as if it was annotated with `e_uninit()`. The issues related to resource initialization will be discussed further in Section II-F.

As far as misaligned pointers are concerned, conversions involving pointers are not permitted in C-rusted, with the exception of converting to/from `void*` in order to call or receive return values from functions of the C Standard Library and POSIX library.

The constraints imposed by C-rusted upon different kinds of references provide protection against most of the causes of the creation and use of dangling pointers, but not all of them. C-rusted introduces the concept of *region*: a *region* is a disjunctive set of identifiers, each of which denotes a set of resources. Each reference is, implicitly or explicitly, associated to a region: this denotes all the resources that are possibly reachable using that reference. One of the jobs of the *C-rusted Analyzer* is to compute the regions of all references anytime they are used to ensure that all the resources within the respective regions have not been released.

Given that the analysis performed by the *C-rusted Analyzer* is intraprocedural only, some explicit information about regions may be required. An example is given in Figure 10: the `max()` function that returns a shared reference could, in principle:

- always returns a reference to the region of formal parameter `p1`; or
- always return a reference to the region of `p2`; or
- return a reference to one of them or the other, as it is the case.

Using region identifiers as arguments to `e_excl()` and `e_shar()` annotations enable the static analyzer to report all cases where the use of a reference is possibly unsafe because

a released resource is within its region, as it happens at line 17 where the region of `ptr` is $\texttt{r1} \cup \texttt{r2} = \{ \texttt{x}, \texttt{y} \}$.[9]

```
1  #include <crusted.h>
2
3  T * e_shar(r1, r2) max(T * e_shar(r1) p1,
4                         T * e_shar(r2) p2) {
5    if (*p1 > *p2)
6      return p1;
7
8    return p2;
9  }
10
11 void foo(T x) {
12   T *ptr;
13   {
14     T y = 7;
15     ptr = max(&x, &y);
16   }
17   printf("%d\n", *ptr);
18 }
```

Fig. 10. Regions

The *C-rusted Analyzer* performs type inference also for regions; for example, if a function returns an exclusive reference and among formal parameters there is only one exclusive reference, then their region is assumed to be the same and no annotation is required. Furthermore, note that the *C-rusted Analyzer* checks, for each use of a reference, not only that all the resources within its region are live but also, for those of them that are references, that they are usable. Regions, together with the ownership model and borrowing, ensure temporal memory safety.

E. Out-of-Bounds Accesses

An out-of-bounds access occurs when a reference to a memory buffer is used to access (for reading or writing) a memory location outside of the intended buffer boundaries.

Out-of-bounds accesses are one of the most subtle and dangerous issues for both software and hardware.[10] In fact,

- out-of-bounds accesses are difficult to spot and they can corrupt memory in an unpredictable way;
- they can be exploited to cause system crashes, to access private information, and to alter the program control flow to the point of executing arbitrary code.

In C, a program performing out-of-bounds accesses has undefined behavior: for the sake of efficiency, C does not require run-time checks to be performed to ensure safety of memory accesses.

The solution adopted in C-rusted is to explicitly annotate references with bound annotations, so that they become *iterators* and allow the *C-rusted Analyzer* to statically check that all accesses are within the correct bounds. In all cases where such guarantees cannot be provided at compile time, a warning is reported that can be addressed by the programmer by inserting the appropriate bound checks.

A reference that cannot be used to iterate the referred resource (i.e., it cannot be incremented or decremented) is said to be an *non-iterable reference*. Examples of non-iterable references are all owning references as well as all references for which the bounds of the referred resources are not known.

A *forward iterator* is an exclusive or shared reference that enables the detection of whether the one-past-the-end element has been reached. This can be achieved by annotating the reference with one of the following bound annotations:

- `e_size()`, which expect as argument an integer variable representing the number of bytes between the referred element and the one-past-the-end element;
- `e_count()`, which expects as argument an integer variable representing the number of elements between the referred element and the one-past-the-end element;
- `e_end()`, which expects as argument a reference to the one-past-the-end element.

Therefore, a forward iterator can be used to forward iterate until the one-past-the-end element is reached.

Figure 11 illustrates two examples of declaration and use of forward iterators.

```
1  #include <crusted.h>
2
3  void foo(T * e_count(n) p, size_t n) {
4    for (size_t i = 0; i < n; ++i) {
5      p[i] = /* ... */;
6    }
7  }
8
9  void bar(T * e_end(end) begin, T *end) {
10   while (begin != end) {
11     // Do something with `begin'.
12     ++begin;
13   }
14 }
```

Fig. 11. Forward iterators

A *bidirectional iterator* is an exclusive or shared reference that enables the detection of whether the beginning element or the one-past-the-end elements have been reached. Such information is provided by passing two identifiers to the bound annotations, denoting the information regarding the beginning and the past-the-end element, respectively.

C-rusted also offers *string iterators*: these are exclusive or shared references having, possibly const-qualified or restrict-qualified `char*` type, and either they definitely refer to a null-terminated string or are formal parameters. Such iterators can be used to forward iterate the string until the null character is encountered. Note that the invariance of the null termination

[9] Rust has something similar to the region annotations: these are called "lifetime annotations."

[10] See, e.g., https://cwe.mitre.org/data/definitions/119.html

shall be preserved, for otherwise a warning will be reported as shown in Figure 12.

```
#include <crusted.h>

void foo(char *s, char c) {
  while (*s != '\0') {
    // Do something with `s'.
    ++s;
  }

  *s = c;
}
```

Fig. 12. String iterator with overwriting of the terminator

```
#include <crusted.h>

void elem_init(T * e_init() p);

void vec_init(T * e_init() e_count(n) vec,
              size_t n) {
  for (size_t i = 0; i < n; ++i) {
    if (SOME_CONDITION)
      elem_init(&vec[i]);
    else
      vec[i] = i;
  }
}
```

Fig. 13. Resource initialization

The problem of detecting out-of-bounds accesses through static analysis is an undecidable problem. Bound annotations can be exploited by the static analyzer to improve the analysis precision, thus obtaining guarantees of spatial memory safety without sacrificing performance for a wider range of situations. When the code is particularly convoluted and the *C-rusted Analyzer* is not able to prove the absence of out-of-bounds accessed despite bound annotations, then the required bound checks can be explicitly added in the code, so that the static analyzer can exploit them to provide safety guarantees: in those cases, a little bit of run-time overhead will be the price to pay to obtain spatial memory safety.

F. Reading Uninitialized Memory

In C it is possible to have uninitialized memory: this is memory whose value is *indeterminate* until it is written for the first time. Reading uninitialized memory is undefined behavior.

C-rusted manages uninitialized memory similarly to optional types: every formal parameter is considered as initialized by default, meaning that whenever possibly uninitialized memory is passed to or returned from a function, this shall be made explicit via a `e_uninit()` annotation. This is what happens, implicitly, for the return type of the `malloc()` function. This requires the support for *initialization functions*: the `e_init()` annotation is used to annotate formal parameters so as to specify that the function will completely initialize the referred resource (i.e., it will write it completely and not read any part of it without prior writing).

Two initialization functions are depicted in Figure 13: note that the initialization reference is an exclusive reference and, in the case of arrays, bound information can be provided so that the reference becomes a forward iterator capable of initializing the whole array.

III. 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* [4]: a Boolean is not an integer, even when it is represented by an `int`, as it may be the case in C90 implementations [6], [7]. Similarly, an object of enumerated type is not an integer, despite being represented by an implementation-defined integer type. Generally speaking, nominal typing allows to impose 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.

```
typedef int e_type() fd_t;
typedef fd_t e_own() fd_own_t;
typedef fd_own_t e_opt(-1) fd_opt_own_t;

fd_opt_own_t open(const char *path,
                  int oflag);

int close(fd_opt_own_t e_release() fildes);
```

Fig. 14. 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 14 shows how some functions involving file descriptors are interpreted in 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 one to the other, and operations that are permitted on integers are not permitted on file descriptors.

It is also possible to use global annotations `e_uop()` and `e_bop()` to define, respectively, unary and binary *nominal operations*: a method to explicitly permit or deny operations on the types given as arguments.

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: as an example, nominal typing can prevent accidentally mixing different unit of measure in a program that manipulates physical entities, independently of the underlying C data types.

IV. Safe and Unsafe Boundaries

Sometimes, the need may arise to use some of the features of the C programming language without submitting to the constraints imposed by C-rusted. This can happen in the implementation of some libraries, in the development of low-level systems where access to the hardware is essential or in peculiar portions of a program where the code is particularly convoluted and obscure; for such cases, the static analyzer may be afflicted by false positives. For all these situations, a feasible solution might be to temporary "escape" some of the constraints imposed by C-rusted: `e_unsafe()`, `e_unchecked()` and `e_checked()` annotations allow this.

`e_unsafe()` annotation 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." This also can be used to enforce information hiding and a sharp separation between interface and implementation by means of a flexible access restriction system.

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 pointers 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` [4, Rule 22.5]. Figure 15 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*. Note that the string literal argument in `e_unsafe()` is arbitrary, which allows an unlimited number of "unsafety kinds."

For the implementation side, C-rusted provides two annotations: `e_unchecked()` and `e_checked()`. The first marks a statement as not expected to conform to the C-rusted safety and security requirements: every function containing unchecked statements shall thus be annotated as unsafe. The latter 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 16 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 6. As a result, under

```
1 e_decl_props(FILE, e_unsafe("FILE"));
2
3 #define e_opt_own() e_opt() e_own()
4
5 FILE * e_opt_own()
6 fopen(const char * restrict filename,
7       const char * restrict mode);
8
9 int fclose(FILE * e_own() e_release() fp);
```

Fig. 15. C-rusted view of `fopen()` and `fclose()`

```
1  #include <crusted.h>
2
3  int fclose(FILE * e_own() e_release() fp) {
4    // ...
5
6    e_checked("FILE") {
7      if (fp->flags == 0U) {
8        errno = EBADF;
9        return EOF;
10     }
11   }
12
13   // ...
14 }
```

Fig. 16. Fragment of `fclose()` implementation with C-rusted annotations

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.

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.

V. 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* [8]. 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. Relational constraints among variables are tracked using the *Parma Polyhedra Library* (PPL) [9]. 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 [2, 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.

VI. Conclusion

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 [10].

For C, a similar approach is the one of *Checked C* [11]. 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: the compilation of Checked C code requires a special compiler.

C-rusted is not a new programming language: C-rusted code is standard ISO C code just used in a peculiar way and in association with suitable static analysis techniques. 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. For instance, C-rusted is 100% compatible with MISRA C: a C program that is MISRA compliant can be annotated 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 [12] prescribe the use of safe subsets of standardized programming languages used with qualifiable translation toolchains (see, e.g., [13] and [14]). 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 and any qualified C compiler is, as is, a qualified C-rusted compiler.

References

[1] D. Keaton. (2020, Nov.) Programming language C — C23 Charter. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2611.htm

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

[3] White House Office of the National Cyber Director. (2024, Feb.) Back to the building blocks: A path toward secure and measurable software. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[4] 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.

[5] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2022, online version available at https://doc.rust-lang.org/book/ and maintained at https://github.com/rust-lang/book, last accessed on March 9, 2024.

[6] ISO/IEC, *ISO/IEC 9899:1990: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1990.

[7] ——, *ISO/IEC 9899:1990/AMD 1:1995: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1995.

[8] 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.

[9] R. Bagnara, P. M. Hill, and E. Zaffanella, "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Science of Computer Programming*, vol. 72, no. 1–2, pp. 3–21, 2008.

[10] 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

[11] 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

[12] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.

[13] ——, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 8: Supporting processes*. Geneva, Switzerland: ISO, Dec. 2018.

[14] RTCA, SC-205, *DO-330: Software Tool Qualification Considerations*. RTCA, Dec. 2011.