

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 economical 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 usage, 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 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 [2], 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 [3]. 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 [4], [5], [6], [7], [8], [9], [10], [11]. 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

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 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 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 diagnostics, summarized in Figure 2, where the notation w_n decorating a program point means that the indicated warning is given at that program point.

A much saner version of the program is depicted in Figure 3. This makes it clear that, while the (static) type of `fd` is `int` for its entire lifetime, 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 *dynamic properties* of objects.

Figure 3 allows us making some observations. First, 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 commonly used library). Secondly, the annotation is not heavy and does not clutter the code. Type qualifiers, such as `e_hown`, can also be embedded in typedefs and a proper choice of typedef names also helps readability and understandability.

The annotation language allows expressing constraints on the use of *resources* via *handles*. 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 two kinds of resources: (1) memory and (2) any language-defined, system-defined or user-defined abstraction with a definite life cycle. A *handle* is any C expression that is able to refer to a resource. Pointers and file descriptors are examples of handles.

C-rusted distinguishes between different kind of handles:

Owner handles: An *owner handle* referring to a resource has a special association with it. In a safe C-rusted program,² every resource subject to explicit disposal (as opposed to implicit disposal, as in the case of stack variables going out of scope), must be associated to one (and only one) owner handle. Through the program evolution, the owner handle for 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 current owner and the owned resource only ends when a designated function is called to dispose of the resource. Note that an owner handle is a kind of *exclusive handle*.

¹The 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.

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

```

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

#define BUFSIZE (100U)

extern void process(char *string);

int main(int argc, const char *argv[]) {
    if (argc != 2)
        return 1;

    int fd = open(argv[1], 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

```

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

#define BUFSIZE (100U)

extern void process(char *string);

int main(int argc, const char *argv[]) {
    if (argc != 2)
        return 1;

    int fd = open(argv[1], O_RDONLY);
    char *buf = (char *) malloc(BUFSIZE);
    ++fdw1;
    ssize_t bytes = read(fdw2, bufw3, BUFSIZE - 1U);
    buf[bytes]w4w5 = '\0';
    process(bufw6);
    returnw7w8 0;
}

```

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/must it deallocate its argument?

w_7 : The file descriptor contained in `fd` is definitely leaked here.

w_8 : The memory pointed to by `buf` is possibly leaked here.

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

```

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

#define BUFSIZE (100U)

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

int main(int argc, const char *argv[]) {
    if (argc != 2)
        return 1;

    int fd; // `fd` value is indeterminate.
    fd = open(argv[1], O_RDONLY);
    // `fd` value is either the erroneous value -1 or an open file descriptor.
    if (fd == -1)
        return 1;

    // `fd` value is definitely an open file descriptor.

    char *buf = (char *) malloc(BUFSIZE);
    // `buf` value is either null or points to a heap-allocated char array.
    if (buf == NULL)
        return 1;

    // `buf` value definitely points to a heap-allocated char array.
    ssize_t bytes = read(fd, buf, BUFSIZE - 1U);
    // `bytes` value is either the erroneous value -1 or the number of bytes
    // read into `buf`.
    if (bytes == -1)
        return 1;

    // `bytes` value is definitely the number of bytes read into `buf`.
    buf[bytes] = '\0';
    // process() takes ownership of `buf` and will deallocate it: no memory leak.
    process(buf);

    // close() properly closes the file descriptor contained into `fd`:
    // no file descriptor leak.
    close(fd);

    // `fd` value is an ordinary integer and cannot be used as a file descriptor
    // (it can be overwritten of course).
    // ...

    return 0;
}

```

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

Exclusive handles: An *exclusive handle* referring to a resource also has a special association with it: while the resource cannot be disposed via an exclusive non-owner handle (only an owner handle allows that), the exclusive handle allows modification of the resource. As a consequence of this fact, no more than one usable exclusive handle may exist at any given time; moreover, the existence of an usable exclusive handle is incompatible with the existence of any other usable handle;

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

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

Let us consider nominal typing. It is already used, e.g., in the MISRA C *essential type model* [12]: a Boolean is not an integer, even when it is implemented 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. An example that is similar in spirit to the one of file descriptors concerns the use of `FILE` pointers in the C Standard Library. The application programmer ought to treat those 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 would be prevented, but there is no such a guarantee. In fact, MISRA C:2012 has a mandatory rule that bans dereferencing pointers to `FILE`, but the rule does not prevent them, e.g., being incremented or being arguments of the `>=` relational operator [12, Rule 22.5]. This is why in C-rusted, the `fopen()` standard function is treated as if it was declared by

```
typedef FILE* e_type e_std_FILE_handle;
e_std_FILE_handle e_opt(0) e_own
    fopen(const char *p, const char *m);
```

where `e_opt(0)` means that the returned pointer may be null and `e_own` means that it has ownership. 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.

A different but related concept is the one of *nominal subtyping* whereby the subtype inherits part of the properties from its representation. For instance, in C-rusted you can write

```
typedef float e_sub celsius_t;
typedef float e_sub kelvin_t;
inline kelvin_t
celsius_to_kelvin(celsius_t c_deg) {
    // ...
}
```

This, along with the ban of casts involving nominal types, will prevent accidentally mixing temperature scales, independently from the underlying C data types. It is also possible to specify which operations are admitted and what is the result. For instance, while `celsius_t - celsius_t` is admissible as is `kelvin_t - kelvin_t` and these give `celsius_t` and `kelvin_t`, respectively, `celsius_t / celsius_t` must be flagged whereas `kelvin_t / kelvin_t` makes perfect sense. Nominal subtyping has other important applications, such as keeping a sharp distinction between data that is possibly tainted or confidential and data that is not.

Finally, let us consider C-rusted ability to capture the way in which functions modify the dynamic properties, including user-defined properties, of all sorts of resources. For example, a user-defined kind of mutex can be associated — in addition to the default C-rusted properties for resources being uninitialized (when they have been allocated but not initialized), initialized (when they have been initialized) or dead (when they have been disposed)— with a *locked/unlocked* dynamic property. The lock function is annotated to require an exclusive reference to a mutex in *unlocked* state and will provoke a *unlocked-to-locked* transition, and dually for the unlock function. The function destroying mutexes is annotated to require an owner handle to a mutex in *unlocked* state. All deviations from these requirements are flagged as violations by the *C-rusted Analyzer*.

III. 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 and can be taught to programmers in a week of training.

Only when the addition of annotations shows the presence of a problem will a code modification be required in order to fix the latent bug that is now visible: in all other cases, the code behavior will remain exactly the same.

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

	C	C-rusted	Rust
Standardized	Yes: ISO	Yes: it <i>is</i> ISO C	No: moving target
Certifiable translators exist	Yes	Yes: it <i>is</i> 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. 4. Advantages and disadvantages of C-rusted (along with its C inheritance) and Rust

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 to write more code in the significantly less secure and verifiable JavaScript language [15].

Figure 4 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.³ 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.⁴ 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 [19] prescribe the use of safe subsets of standardized programming languages used with qualifiable translation toolchains (see, e.g., [20] and [21]). 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 is 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 dynamic properties of objects 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, 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., file descriptors 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 dynamic 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 a honorable operational

³C-rusted is compatible with any version of the ISO C Standard and can be used with any C toolchain.

⁴We note on passing that, in the authors' opinion, C-to-Rust transpilation [16], [17], [18] 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.

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 handles.

IV. 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* [22]. 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, field-sensitive (context-insensitive) points-to analysis and a domain for the tracking of numeric information. In addition, there are several finite domains specifically conceived for C-rusted, which track the state of resources and handles as well as the evolution of dynamic object properties. Scalability is ensured by intraprocedural analysis.

All the annotations of C-rusted are realized via macro invocations: the corresponding macros all expand to the empty token sequence 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] 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.
- [3] 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>
- [4] 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/>
- [5] B. Cantrill, “Is it time to rewrite the operating system in Rust?” *InfoQ*, Jan. 2019. [Online]. Available: <https://www.infoq.com/presentations/os-rust/>
- [6] J. Wallen, “Let the Linux kernel Rust,” *TechRepublic*, Jul. 2021. [Online]. Available: <https://www.techrepublic.com/article/let-the-linux-kernel-rust/>
- [7] 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/>
- [8] —, “Where rust fits into linux,” *The Register*, Nov. 2021. [Online]. Available: https://www.theregister.com/2021/11/10/where_rust_fits_into_linux/
- [9] 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/>
- [10] 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/>
- [11] N. Elhage, “Supporting Linux kernel development in Rust,” *LWN.net*, Aug. 2020. [Online]. Available: <https://lwn.net/Articles/829858/>
- [12] MISRA, *MISRA-C:2012 — Guidelines for the use of the C language critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2019, third edition, first revision.
- [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->
- [16] 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.
- [17] 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.
- [18] 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.
- [19] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.
- [20] —, *ISO 26262:2018: Road Vehicles — Functional Safety — Part 8: Supporting processes*. Geneva, Switzerland: ISO, Dec. 2018.
- [21] RTCA, SC-205, *DO-330: Software Tool Qualification Considerations*. RTCA, Dec. 2011.
- [22] 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.