The Challenge of using C in Safety-Critical Applications

Shea Newton, Nathan Aschbacher
PolySync Technologies, Inc.

Abstract

Software errors in safety-critical systems can have severe consequences: property-loss, environmental devastation, injury, or death. Despite the severity of these risks, software continues to be written for safety-critical applications in languages that permit common classes of failures, such as undefined behavior, state corruption, and unexpected termination. One such language is C. Language standards that define allowable subsets (e.g. MISRA) and static analysis tools are often used in an attempt to ameliorate these failures by detecting them in the program code before they result in a critical issue at runtime. These traditional methods are ultimately insufficient when it comes to providing ahead-of-time assurances about safe runtime behavior for safety-critical applications. Alternative approaches must be considered.

Introduction

The cost of software errors in safety-critical applications carry the potential for physical injury and death.[1] Software errors also carry a quantifiable monetary cost. In 2002 it was estimated at $59.5 billion annually to the U.S. economy.[2]

Strategies for reducing these costs by catching software errors early in the development process have evolved to identify patterns known to be unsafe or prone to error.[3] Among those strategies, thorough design documents and regular code audits are generally constants in today's software development processes, but they are susceptible to the same vulnerability as a software's implementation: human error. Static code analysis is used to mitigate this issue by automating the process of catching dangerous patterns and practices.[4]

Background

The presence of C in safety-critical systems is near-ubiquitous. Among the many reasons for its prevalence are performance, control, footprint, and compiler support. However, C also has very permissive semantics which can make it dangerous. A draft of the 2010 ISO/IEC 9899:2011 standard details nearly two-hundred scenarios for undefined behavior.[5] Still, that ubiquity means it is often the most likely candidate even for new safety-critical software development.

The permissiveness of the C standard has given rise to other standards that promote only subsets of functionality, limiting the potential for undefined behavior. Some notable examples include: The Computer Emergency Response Team (CERT) developed CERT C and CERT C++, Lockheed Martin's JSF++ AV, and the Barr Group promoted Netrino C. For applications in the automotive field a commonly applied standard is MISRA-C.

These subsets of the C standard are often promoted as a path to safer software. The implication being that some safer subset equates to being good enough for safety-critical use cases.[6]

This paper seeks to explore common runtime errors to determine whether *safer*, as provided by standardized language subsets and/or static analysis tools, is a meaningful distinction for the C language.

Significance Argument from a Constrained Space of Runtime Errors

MISRA-C requires that all `switch` statements have a `default` case and that all `if ... else if` statements are terminated with an `else`. It does not necessarily follow that this kind of constraint provides any assurances about reduction of possible runtime errors.[7] In order to make claims about constraints on the space of possible runtime errors, we need complete requirements. For example, that all cases are accounted for.

[1] Wikipedia contributors. (2018) 2009–11 Toyota vehicle recalls. Available at: https://en.wikipedia.org/wiki/2009-11_Toyota_vehicle_recalls [Accessed April 11, 2018]

[2] The National Institute of Standards and Technology(NIST), Section ES.6 of the Planning Report 02-3. Available at: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf [Accessed April 11, 2018]

[3] Fagan M.E. (2002) Design and Code Inspections to Reduce Errors in Program Development.

[4] Wichmann B.A., Canning A.A, Clutterbuck D.L., Winsborrow L.A., Ward N.J, Marsh D.W.R (1995) Industrial perspective on static analysis.

[5] ISO/IEC 9899:201x (Committee Draft, 2010). Available at: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf [Accessed April 11, 2018]

[6] JSF AV C++. http://www.prqa.com/coding-standards/jsf-av-c/ [Accessed April 11, 2018]

[7] Boogard C, Moonen L (2008) Assessing the Value of Coding Standards: An Empirical Study.

Significance Argument from the Cost of Catching Bugs

The cost of software errors extends to the time and effort required to diagnose, document, and remedy discovered issues. Because these issues are likely unpredicted, the time and effort required is often unaccounted for, requiring that other in-progress tasks be postponed. Remediation comes with the risk of introducing other issues, more bugs created in rework and redesign, and technical debt due to urgency around a fix. Catching software errors at compile-time or before runtime means saving, money, time, and potentially lives.

Significance Argument from the Feasibility of the Problem

In languages like C the problem of comprehensive static analysis may be NP complete.[8] The implication of that is static analysis of C code is limited to an approximation; there will always be potential for gaps in the capability of the tools.

Methods

To evaluate what assurances the MISRA-C standard, and C-oriented static analysis tools are capable of, we selected four classes of dangerous behaviors permitted by the C language standard: multiple mutable aliases, modification of immutable data, ambiguous pattern matching, and data races. We then applied the MISRA-C guidelines and several C-oriented static analysis tools to make a determination of how effective those practices and tools were in exposing dangerous sources of software failures in code.

We also investigated multiple additional standards as potential complements to MISRA-C, but best-practice guidelines across standards were generally in direct conflict with each other; making it difficult to produce a compliant program at all. Examples of such conflicts included:

- Using `#define` the keywords `goto` and `continue` to raise compiler errors if used.
- No reserved words can be redefined with `#define`.
- `#define` must not be used.

Additionally, style-related issues like variable casing and whitespace requirements varied regularly between standards. Because of these conflicts and because MISRA-C is a de facto standard in the automotive space, the scope of the results published in this paper are limited to MISRA-C.

We selected a leading proprietary compliance tool for its industry acceptance, and reputation as state of the art to check for MISRA-C violations. Though that suite is able to check against a broad set of standards, we opted instead to evaluate a diversity of tooling to ensure that a single origin could not account for any of the results shown. The additional set of tooling included Cppcheck,[9] Flawfinder,[10] Flint++,[11] Frama-C,[12] OCLint,[13] scan-build,[14] splint,[15], and Vera++.[16]

In the vein of a proof by contradiction, we started with the premise that some set of static analysis tooling is able to guarantee safety from a given error caused by one of the classes of dangerous behavior enumerated previously. Next, we wrote a simple program capable of representing each class of error and recorded the violations reported by each of the static analysis tools.

The example source cited in this paper has been modified for illustrative purposes but is available for reproducibility as a `static-analysis-argumentation` repository on GitHub.[17]

Mutable Aliasing

Where multiple mutable aliases are permitted there will always be a risk of inadvertently using an invalid reference. To combat this MISRA-C requires runtime checks for NULL pointers. Some static analysis tooling performs very well when warning about code that looks like dereferencing a NULL pointer. Other types of data corruption however, are less detectable.

The following is a reduced version of the sample program written to violate assurances about memory safety the MISRA-C standard, or the static analysis tooling we evaluated, might be used to make.

The program creates a reference to valid data. Next, it aliases that reference and corrupts it. Afterward, using the reference results in undefined behavior, generally a segmentation fault.

```
typedef struct {
    int32_t x;
    int32_t y;
    int32_t z;
} example_s;
/* Example data. */
```

[8] Landi W.(1992) Undecidability of static analysis.

[9] Cppcheck. Available at: https://github.com/danmar/cppcheck [Accessed April 11, 2018]

[10] Flawfinder. Available at: https://www.dwheeler.com/flawfinder [Accessed April 11, 2018]

[11] Flint++. Available at: https://github.com/JossWhittle/FlintPlusPlus [Accessed April 11, 2018]

[12] Frama-C. Available at: http://frama-c.com [Accessed April 11, 2018]

[13] OCLint. Available at: https://github.com/oclint/oclint [Accessed April 11, 2018]

[14] scan-build. Available at: https://clang-analyzer.llvm.org/scan-build.html [Accessed April 11, 2018]

[15] splint(1) - Linux man page. https://linux.die.net/man/1/splint [Accessed April 11, 2018]

[16] Vera++. Available at: https://bitbucket.org/verateam/vera/wiki/Introduction [Accessed April 11, 2018]

[17] static-analysis-argumentation. Available at: https://github.com/PolySync/static-analysis-argumentation [Accessed April 11, 2018]

```
example_s a = {
    .x = 1,
    .y = 2,
    .z = 3
};
/* Mutable reference. */
example_s * b = &a;
/* Mutable alias. */
example_s ** c = &b;
/* Reference corrupted. */
*c += 2048;
/* Use after corruption. */
b->y = 4;
b->x = 5;
b->z = 6;
```

The following table represents the violations resulting from static analysis of the source file `alias.c` available on GitHub.[18]

Tool	Violations Reported
MISRA-C Checker	-
Cppcheck	-
Flawfinder	-
Flint++	Prefer 'nullptr' to 'NULL'
Frama-C	-
OCLint	-
scan-build	-
splint	'main' should return 'int'

Table 1: Results of the static analysis of `alias.c`

The premise of mutable aliasing is at the root of all the other errors we explore. This base case is exploited again in each of the following examples in order to provide loud results in the form of segmentation faults.

Breaking the 'const' Promise

In an attempt to prohibit the previous behavior, this example illustrates a case where even an attempt to make a reference immutable is a contract that can be broken. This program again creates a reference to valid data, but this time the reference has a constant qualification. That reference is aliased and corrupted despite the qualification. When the reference is used later, it results in undefined behavior; generally a segmentation fault.

```
/* Example data. */
example_s a = {
    .x = 1,
    .y = 2,
    .z = 3
};
/* Constant reference to
* example data.
*/
const example_s const * b = &a;
/* Constant alias to reference. */
const example_s * const * c = &b;
/* Cast away const to corrupt
* reference. */
*((example_s **)(c)) += 2048;
/* Use reference after
* corruption. */
int32_t sum = b->a + b->b + b->c;
```

The following table represents the violations resulting from static analysis of the source file `constant.c` available on GitHub.[19]

Tool	Violations Reported
MISRA-C Checker	-
Cppcheck	-
Flawfinder	-
Flint++	Prefer 'nullptr' to 'NULL'
Frama-C	-
OCLint	-
scan-build	-
splint	'main' should return 'int'
Vera++	-

Table 2: Results of the static analysis of `constant.c`

Unreliable Pattern Matching

There are no safeguards from misusing enumerations in C. They have the look and feel of a new type but are indistinguishable from integer constants in use. In this example an enum is matched on an unrelated value and leads, once again, to the use of corrupted data.

```
/* Enumeration intended for use. */
typedef enum {
APPLY_BRAKE = 1,
APPLY_THROTTLE = 2
} action_e;
/* Ambiguous enumeration. */
enum {
SELF_DESTRUCT = 2,
};
/* Example data. */
example_s a = {
    .x = 1,
    .y = 2,
    .z = 3
};
/* Mutable reference. */
example_s * b = &a;
/* Mutable alias. */
example_s ** c = &b;
/* Indended use as an 'action_e'
```

[18] static-analysis-argumentation. Available at: https://github.com/PolySync/static-analysis-argumentation/blob/master/c_examples/alias.c [Accessed April 11, 2018]

[19] static-analysis-argumentation. Available at: https://github.com/PolySync/static-analysis-argumentation/blob/master/c_examples/constant.c [Accessed April 11, 2018]

```
* enum type.
*/
action_e t = APPLY_THROTTLE;
/* Match on integer instead. */
switch (t)
{
    /* Wrong pattern. */
    case SELF_DESTRUCT: {
    *c += 2048; break;
    }
    default: { break; }
}

/* Use after corruption. */
b->y = 4;
b->x = 5;
b->z = 6;
```

The following table represents the violations resulting from static analysis of the source file `pattern.c` available available on GitHub.[20]

Tool	Violations Reported
MISRA-C Checker	-
Cppcheck	-
Flawfinder	-
Flint++	Prefer 'nullptr' to 'NULL'
Frama-C	-
OCLint	-
scan-build	-
splint	'main' should return 'int'
Vera++	-

Table 3: Results of the static analysis of `pattern.c`

Data Races

This example uses a data race condition to corrupt its reference to a shared resource. Taken in an asynchronous context, the corrupting logic below is unreachable. Inside a threaded callback however, corruption of the shared resource occurs reliably.

```
/* Arbitrary bound. */
while ((b != NULL) && (b->x < 10))
{
    /* Simulate some amount of
    * work.
    */
    (void)sleep(0);

    /* If another thread has
    * changed the shared resource.
    */
    if (b->a >= 10)
    {
        b += 2048;
    }
    else
    {
        /* Increment potentially
        * corrupted reference.
        */
        b->x += 1;
    }
}
```

The following table represents the violations resulting from static analysis of the source file `thread.c` available on GitHub.[21].

Tool	Violations Reported
MISRA-C Checker	1. Static procedure is not explicitly called. 2. Casting operation on a pointer
Cppcheck	-
Flawfinder	-
Flint++	Prefer 'nullptr' to 'NULL'
Frama-C	'_WORDSIZE' redefined
OCLint	-
scan-build	-
splint	Parse Error: Non-function declaration
Vera++	-

Table 4: Results of the static analysis of `thread.c`

A Different Approach

We want to make assurances about the behavior of our safety-critical software. After setting out to discern whether the MISRA-C standard and static analysis tooling could provide us those assurances, we were able to contradict that possibility in several cases. An inability to reliably identify faults injected into simple programs–using the static analysis tools identified in this paper–diminishes the confidence that we could reliably discover unintentional faults in complex programs.

Of the potential approaches to providing safety assurances, we opted to explore other programming languages. Many offer protection from the class of errors we detailed. A few of the outstanding candidates were Haskell, SPARK Ada, and Rust.

For the purposes of this paper we chose the Rust language to illustrate that our selected error scenarios are preventable before runtime.

The following examples are reduced for brevity but are also available in the `static-analysis-argumentation`

[20] static-analysis-argumentation. Available at: https://github.com/PolySync/static-analysis-argumentation/blob/master/c_examples/pattern.c [Accessed April 11, 2018]

[21] static-analysis-argumentation. Available at: https://github.com/PolySync/static-analysis-argumentation/blob/master/c_examples/thread.c [Accessed April 11, 2018]

repository on GitHub.[22]

Rust disallows much of the behavior C permits. The method here represents idiomatic Rust more so than literal translations from C. For example, we opted to use the `Box` type over `*mut`. Attempting to dereference a `*mut` without an `unsafe` block also results in compilation errors, but does not illustrate the fundamental differences between each language's approach.

Safe Aliasing

The following example does not compile and the attempt to corrupt our reference fails. Rust's concept of borrowing and ownership saves us here. Though we are able chain together mutable references to our data—each one borrowing ownership from the other—we are not able to destroy what we don't own.

```
struct Example {
    x: i32,
    y: i32,
    z: i32,
}
let mut a: Example =
    Example { x: 1, y: 2, z: 3 };
let mut b: Box<&Example> =
    Box::new(&mut a);
let _c: Box<&&Example> =
    Box::new(&mut b);
drop(**c);
//~^ ERROR cannot move out of
// borrowed content
b.x = 4;
//~^ ERROR cannot assign to `b.x`
// because it is borrowed
b.y = 5;
//~^ ERROR cannot assign to `b.y`
// because it is borrowed
b.z = 6;
//~^ ERROR cannot assign to `b.z`
// because it is borrowed
```

The Immutable Promise

In this example, we take the same approach but with constants for extra effect. The additional errors shown below are due to the attempt to mutate immutable data. In Rust, the const promise isn't so easily broken.

```
let mut a: Example =
    Example { x: 1, y: 2, z: 3 };
let mut b: Box<&Example> =
    Box::new(&mut a);
let c: Box<&&Example> =
    Box::new(&mut b);
// Attempt to corrupt referenced data
drop(*(&mut(**c)));
//~^ ERROR cannot borrow immutable
// `Box` content `*c` as mutable
b.x = 4;
//~^ ERROR cannot assign to field
// `b.x` of immutable binding
b.y = 5;
//~^ ERROR cannot assign to field
// `b.y` of immutable binding
b.z = 6;
//~^ ERROR cannot assign to field
// `b.y` of immutable binding
```

Reliable Pattern Matching

In this example we've opted to `panic!` if the pattern is matched incorrectly. However, this program fails to compile, not even producing a potentially dangerous artifact. The first thing to note is that Rust disallows ambiguous enums. Second, despite the fact that both `ApplyBrake` and `SelfDestruct` are assigned the value `1`, Rust doesn't allow one pattern matched as another by default. It is also worth noting that Rust doesn't require an arbitrary number of patterns in match statements—simply that each pattern is accounted for.

```
// Enumeration intended for use.
enum Action {
    ApplyBrake = 1,
    ApplyThrottle = 2,
}
// Second enumeration.
enum Destruct {
    SelfDestruct = 1,
}
let pattern = Action::ApplyThrottle;
match pattern {
    Destruct::SelfDestruct =>
    //~^ ERROR mismatched types
    {
        panic!("Memory corrupted.")
    }
}
```

Protection from Data Races

We'll take this example in two parts. First, we directly attempt to implement the behavior displayed in the C example of a data race condition. There are many complaints raised by the Rust compiler here around concepts of ownership we saw in previous examples. More explicitly however, this example leverages Rust's concept of move semantics. This code violates the requirement that there is always *exactly one* binding to any given resource.[23]

[22] static-analysis-argumentation. Available at: https://github.com/PolySync/static-analysis-argumentation [Accessed April 11, 2018]

[23] Ownership. Doc.rust-lang.org. Available at: https://doc.rust-lang.org/1.8.0/book/ownership.html [Accessed April 11, 2018]

```
while b.x < 10 {
//~^ ERROR use of moved value: `b.x`
    thread::sleep(
        time::Duration::from_secs(0));
    if b.x >= 10 {
    //~^ ERROR use of moved value:
    // `b.x`
        drop(b);
        //~^ ERROR use of moved value:
        // `b`
    }
    b.x = b.x + 1;
    //~^ ERROR use of moved value:
    // `*b`
}
```

Second, we'll look at the Rust compiler's requirements for sharing data across threads. In the following example, thread setup and teardown have been omitted.

```
let a: Example =
Example {
    x: 1, y: 2, z: 3
};
let b: Box<&mut Example> =
    Box::new(&mut a);
//~^ ERROR `a` does not live long enough
for _ in 0..thread_count {
    handles.push(Some(thread::spawn(
        || { black_box(b)})));
    //~^ ERROR capture of moved value:
    // `b`
}
```

This example illustrates the violation of two of Rust's requirements for thread safety. First, the `spawn` function has what Rust calls a lifetime constraint. `spawn`'s argument, a closure, "must have a lifetime of the whole program execution."[24] Here, because we reference local data, that requirement is not met. If any of the threads spawned were to live longer than the scope of our data, their references would become invalid.

Second, because there is *exactly one* mutable reference to a resource at any given time, our reference must be moved to a thread during the loop's first iteration. Rather than the borrowing behavior we saw previously, moving transfers ownership completely. In the next iteration of the loop, we cannot give what we would no longer own.

Results

The static C code analysis reports generated while developing our examples generally fell into three categories across the tools evaluated.

1. Cosmetic reporting. Errors raised due to inconsistent whitespace, short variable names, lack of copyright notice, etc..
2. Best practice reporting. Recommended minimum `case` declarations in a `switch` statement, trying to return an `in32_t` from a function signature that specifies `uint32_t`, no use of basic types such as `int` or `double`.
3. Danger reports / actual bugs. Catching the use of uninitialized data, dereferencing NULL pointers, out of bounds array indexing.

Most of what is reflected in the C examples that were written are not the bugs that go undetected. The bulk of the code is a requirement of conforming to a tool's requirements. Sometimes addressing static analysis violations meant the addition of significantly more code. Though many of the requirements of that conformity seem intuitively beneficial, modifying a code base to appease the tooling has the potential to make it less maintainable and to introduce other faults.[25]

Tables 1 through 4 exhibit the static analysis violations still present in the source code corresponding to each example. The details of each of those violations are as follows:

Violation: Prefer 'nullptr' to 'NULL'

The `nullptr` keyword is a feature of C++. This violation is an artifact of using Flint++, a C++ linter, on C code.[26]

Violation: 'main' should return 'int's

This violation is the result of conflicting requirements across tooling. MISRA-C directive 4.6 specifies that typedefs indicating signedness and size should be used instead of basic types, as in the following minimal example:

```
typedef signed int SINT_32;
SINT_32 main(void){return 0;}
```

Our approach favored satisfying the requirements of the proprietary MISRA-C checker above the rest of the tooling.

Violation: Static procedure is not explicitly called

The MISRA-C standard does not address callbacks or concurrency. This violation arose because the tool used for MISRA-C checking does not accept that a function passed to `pthread_create` is explicitly used.

Violation: Casting operation on a pointer

This violation does represent an indication of dangerous behavior and a legitimate concern. However, it doesn't

[24] std::thread::spawn - Rust. Doc.rust-lang.org. Available at: https://doc.rust-lang.org/std/thread/fn.spawn.html [Accessed April 11, 2018]

[25] Boogard C, Moonen L (2008) Assessing the Value of Coding Standards: An Empirical Study.

[26] Flint++. Available at: https://github.com/JossWhittle/FlintPlusPlus [Accessed April 11, 2018]

address the root cause of the data race condition the example was written to illustrate. The casting operation from a `void*` is typical in C callbacks and was favored over accessing global data and a violation originating from an unused function parameter.

Violation: '___WORDSIZE' is redefined

This error is the result of Frama-C's analysis of `pthread.h` and originates in the source files outside of the example program `thread.c`.

Violation: Parse-Error: Non-function declaration

This error is the result of splint's analysis of `pthread.h` and originates in the source files outside of the example program `thread.c`.

Conclusion

We have identified clear gaps in the capabilities of static analysis tools for C, as well as in the commonly applied coding standard MISRA-C. Those gaps indicate that the ability to make broad assurances about the safety of software written in C is infeasible using these methods. Given our position that the automotive industry should be pursuing the state of the art and that software written in C with augmentation appears to fall short of that, new opportunities and methods should be investigated for qualification as state of the art.[27]

[27] NI.com (2018) What is the ISO 26262 Functional Safety Standard? - National Instruments. Available at: http://www.ni.com/white-paper/13647/en/ [Accessed April 11, 2018]