Understanding “Memory Safety”

Guarantees, Limits, and Different Solution Approaches

Martin Becker
The MathWorks GmbH
Munich, Germany

Jacob Palczynski
The MathWorks GmbH
Aachen, Germany

***Abstract*—Memory safety has become a trending topic. The severity of memory errors has been recently demonstrated by the CrowdStrike incident, which resulted in more than $5 billion in damages. Authorities and institutions are calling for a transition to memory-safe programming languages and urging the industry to reevaluate their current software development tools. However, while memory safety is essential for software reliability, it is sometimes tragically misunderstood, leading to conclusions that do not necessarily reduce the risk of software failure. This paper aims to demystify memory safety, to challenge oversimplified notions, and present a nuanced perspective on its implementation. We explore both established solutions (e.g., coding guidelines like MISRA C++) and emerging ones (e.g., new languages like Rust) and evaluate their mechanisms and guarantees in the context of embedded systems. We argue that memory safety is not a binary property, but rather a spectrum with many solutions, each having its own tradeoffs. We present a comparative analysis, assessing their impact on development tools, cost, and their fit with current development and certification practices. Overall, we conclude that memory safety cannot be automatically solved with new tools or programming languages; instead, it requires the whole development process to be well-balanced, and thoroughly understood.**

Keywords—embedded software, verification, memory safety

I. Introduction

Discussions on the topic of memory safety are currently trending. In recent years, numerous institutions, including CISA and NSA [1], have expressed deep concern, initiating new directives and research projects [2]. This trend is also evident in Figure 1, which illustrates the search queries on the topic: a historical peak was reached at the end of February 2024, shortly after the White House in the USA issued an official recommendation for a transition to memory-safe programming languages as part of the national security strategy [3].

Memory safety seems new and urgent, but it is an old problem. The first documented discussions can be traced back to 1972 [4] and were prominently demonstrated in 1988 [5]: The "Morris" worm exploited a buffer overflow to infect thousands of computers via the young Internet. Although no direct damage was caused, the power of memory errors became visible for the first time. Over the years, far more significant damage has been

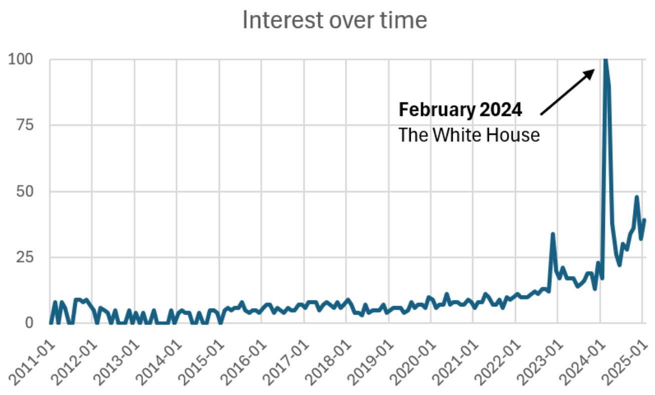


Figure 1: Trending topic of "Memory Safety", data from Google Trends [49].

caused by memory errors, including the recent CrowdStrike incident ($5 billion [6]) and the Heartbleed vulnerability ($500 million [7]). Memory errors are considered one of the most persistent error classes [3] [4], dominating the Top 25 list of the most dangerous software vulnerabilities for many years [8]. Some studies suggest that up to 70% of security vulnerabilities can be attributed to memory errors [9] [10] [11] [12] [13].

In the domain of safety-critical embedded software, the call for memory-safe methods is also growing louder. However, this topic is not new and is already addressed in many regulations (sometimes under different names), such as in functional safety standards like ISO 26262-6 [14, pp. 19,26,46-47] and IEC 61508-3 [15, pp. 27,103]. Moreover, two of the most popular programming languages in embedded systems, C and C++ [16] [17], are considered "memory-unsafe" [1]. Significant effort is required to avoid common memory errors like null pointers and buffer overflows. New memory-safe programming languages like Rust provide new hope, and are slowly being adopted in new projects within the automotive industry [18] [19] [20]. However, many other methods are available, such as memory-safe hardware, compiler extensions, and coding guidelines like MISRA C. How do these differ or complement each other in terms of memory safety, effectiveness, and cost?

```
// memory access at wrong location
int arr[3] = {1, 2, 3};
printf ("%d", arr[3]);
```

```
// memory access at wrong time
int x;
printf ("data is %d\n", x);
```

Figure 2: Spatial (top) and temporal memory error (bottom).

II. Memory Safety – Definition

Memory safety means that no invalid or unintended memory accesses occur in a program. There are two types of memory errors [4] [8], as illustrated in Figure 2:

- Spatial Error: Accesses to unauthorized memory areas or addresses, that is, accessing the wrong memory location. Typical examples include null pointers or writing beyond array bounds (buffer overflows).
- Temporal Error: Accesses to memory at the wrong time. Typical examples are reading uninitialized, incompletely written (race conditions), or already expired (use-after-free) variables.

Impact: Memory errors can cause severe effects, leading to system failures (see CrowdStrike [6]) or security vulnerabilities (see Heartbleed [7]). They can also create behavior that contradicts program logic [21], for example, change the content of unrelated variables or execute the wrong code. In embedded systems, this can result in critical functions malfunctioning or failing, potentially harming users. For instance, driver assistance systems might make unintended steering maneuvers [22], or medical devices might administer incorrect dosages [23]. The specific effects largely depend on the chosen programming language and runtime environment, as we will demonstrate later.

Challenge: Detecting faulty memory accesses is relatively easy (more on this in the next section), but avoiding the underlying causes (prevalence), usually programming or logic errors, is not. The exact control and data flows often depend on input data and can be difficult to predict. Consequently, compilers can only spot memory errors to a limited extent, especially given the high expectations on their speed. Memory safety is often achieved through run-time detection instead of preventing developers from writing faulty code.

Limitation: *It is important to understand that the property "memory safety" merely means that no faulty memory access takes place, but it does not specify how and when it is achieved. Many methods do not prevent developers from writing faulty programs, but merely detect faulty memory access during runtime. Hence, memory safety does not necessarily mean that the underlying logical errors are prevented or remain without impact.* This is a common misconception. Even in a memory-safe programming language, it is possible to write a program that contains logical errors in memory access, and it has consequences. Figure 3 shows an example: A logic error in Rust results in an attempt to read beyond array bounds in line 8. Here, the language guarantees safety by checking the memory access at runtime, recognizing it as faulty, and preventing it through a program termination ("panic"). This reliably protects the program's integrity. An equivalent program in C/C++ could have executed the access unhindered and might have transitioned into undefined and unpredictable behavior [21], likely making it difficult to debug. However, it remains that this terminating behavior is inadequate for embedded software, as it can be seen as a DoS vulnerability according to CWE-125 from the end user's perspective. Instead, the underlying logic errors leading to attempted bad memory access should be identified and corrected.

III. Methods for more Memory Safety

To better assess existing approaches for improving memory safety, we evaluate their impact in two areas: 1. Reducing the frequency or prevalence of underlying errors and 2. Reducing their effects or impact. Additionally, we categorize each approach based on when it is applied (during programming, during compilation, at runtime) and account for the additional runtime overhead (CPU cycles and memory), as well as the costs of initial implementation (reimplementation, training of developers, setup of new tools) and certification (tool qualification, re-work for already certified systems) as suggested in [1] [8]. The results are summarized in Table 1 and explained in more detail below.

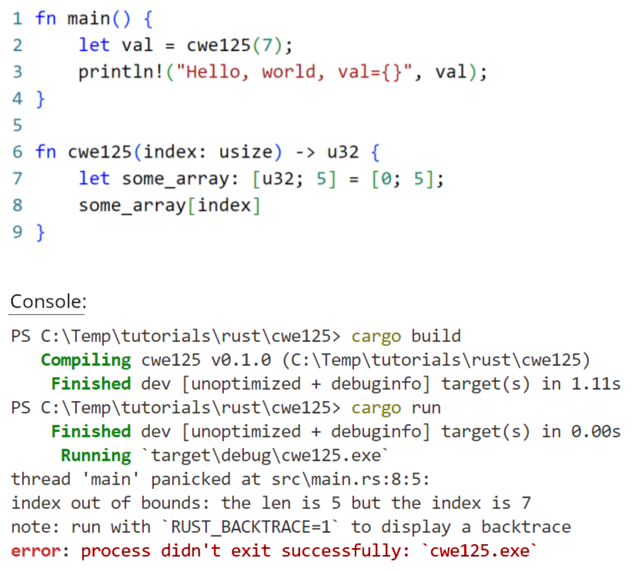
A. Programming Language

Switching to a memory-safe programming language can reduce both the prevalence and impact of memory errors. The degree of improvement depends on the chosen language, but underlying errors and their impacts are never *completely* eliminated.

Memory-safe languages like Rust, Ada, Go, or C# have stricter semantics, enabling more comprehensive checks during compilation and thus uncovering more errors. Typically, a

Table 1: Overview of methods to achieve memory safety.

Method	When			Reduction of Error …		Runtime Overhead	Initial Effort
	Coding	Compilation	Runtime	Prevalence	Impact		
Programming Language	X	X	X	++	++	Low to High	$$$
Coding Guidelines	X			++	+	Low to Medium	$$
Formal Methods	X			+++	o	None	$$
Safe Libraries	X		X	+	+	Medium	$
Compiler Extensions		X	X	+	++	Low to Medium	$
Hardware			X	o	+	Low	$$$
Dynamic Test			X	++	o	High	$$



```
1 fn main() {
2     let val = cwe125(7);
3     println!("Hello, world, val={}", val);
4 }
5
6 fn cwe125(index: usize) -> u32 {
7     let some_array: [u32; 5] = [0; 5];
8     some_array[index]
9 }
```

Console:

```
PS C:\Temp\tutorials\rust\cwe125> cargo build
   Compiling cwe125 v0.1.0 (C:\Temp\tutorials\rust\cwe125)
    Finished dev [unoptimized + debuginfo] target(s) in 1.11s
PS C:\Temp\tutorials\rust\cwe125> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target\debug\cwe125.exe`
thread 'main' panicked at src\main.rs:8:5:
index out of bounds: the len is 5 but the index is 7
note: run with `RUST_BACKTRACE=1` to display a backtrace
error: process didn't exit successfully: `cwe125.exe`
```

Figure 3: Memory-related errors also exist in memory-safe programming languages (here: Rust).

higher level of safety entails a stricter compilation process, which increases development effort but drastically reduces the need for debugging and re-work [8] [24] [25]. Depending on the language, entire defect classes like double-free, race conditions and memory leaks can be eliminated. Additionally, most memory-safe languages rely on runtime checks and exceptions (see Figure 3) to detect faulty memory accesses missed by the compiler, causing runtime overhead. Several studies show that memory-safe languages can significantly reduce the number of memory errors [11]. In Android™, for example, switching to memory-safe languages has reduced memory-related vulnerabilities from 76% to 24% [10].

However, we criticize that some promises go too far. Statements like "Memory errors can simply be *eliminated* by switching to memory-safe programming languages" [1] are, as discussed in the previous section, at best, misleading. Runtime checks, as found in Ada and Rust, are a valid means to ensure memory safety but lead to overhead and, more importantly, program termination, which is an effect inadequate for embedded systems. For example, the CrowdStrike incident [6] would have had the same outcome in a memory-safe programming language since it was a case of missing input validation that can be (and has been by the authors) reproduced in languages like Rust.

Moreover, and also sadly, Memory Safety is often treated as a binary property of a programming language – a language is either memory-safe, or it is not. Instead, it is more helpful to understand it as a spectrum [8] [24]. In principle, safe programs can be written in any language, but certain language features and default behaviors increase the chances of doing so [26]. Even C++, typically categorized as memory-unsafe [1] [3], has memory-safe features like bounds-checking for containers [9] and smart pointers, with more features expected in the future [17] [27] [28]. Similarly, there are language extensions for C that add bounds-checking and improve type safety [8].

Changing the programming language is not easy. In addition to the costs for new tools, training developers, and altering infrastructure, significant costs for (re-)certifications may arise [8] [17] [18]. Applicability may be limited by a lack of qualified personnel and certification risks of new tools. Changing the programming language only applies to newly developed code, while legacy code is not affected. Furthermore, the safety guarantees of languages like Rust may need to be side-stepped when it comes to hardware access, which typically requires the *unsafe* mode. It is notoriously difficult to write robust code in this mode, since the strict language rules still apply, but the compiler can no longer aid with compile-time checks.

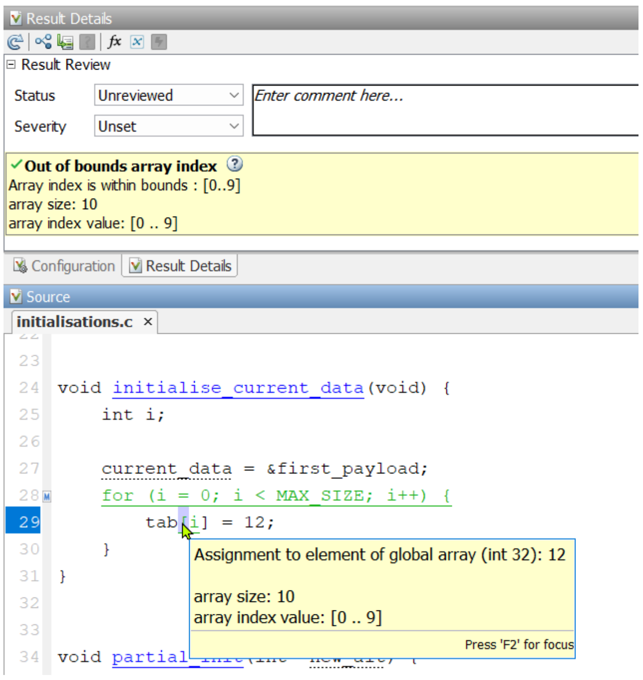
B. *Coding Guidelines and Standards*

Proper use of the programming language plays a crucial role for improving the degree of memory safety and reduce the impact of memory-related errors, regardless of the programming language. Coding guidelines and standards are an effective method for reducing the causes and prevalence of underlying errors. By enforcing certain programming patterns, such as explicit error handling, the impacts of errors are also reduced, typically leading to the use of a "safer language subset." This approach is frequently used in the embedded domain and is well-established in safety standards [1] [15] [14], which strongly suggest the use of coding standards like MISRA C. However, complete error avoidance and elimination of impacts are not possible through this, either.

Established languages have coding guidelines like MISRA C/C++ [17] [29] and static code analysis tools for automatic checking, which sometimes also allow for defining custom rules (e.g., [30]). It is worth emphasizing that coding guidelines are also beneficial for memory-safe languages to prevent errors before runtime and reduce failing run-time checks. As example, Figure 5 shows coding rule that is useful in Rust. The guideline "array accesses must only occur via match expressions" would help to avoid a run-time panic when trying to access an array beyond its bounds. Hence, guideline checkers also exist for the memory-safe languages like Rust [31], but a widely accepted standard like MISRA C/C++ is missing.

An efficient approach to produce code in a safe language subset is Model-Based Design [32]. The code is not written manually but automatically generated from semantic models. The resulting code is well-structured, and – depending on the used code generator and its settings – partially to fully compliant to certain coding guidelines. Moreover, some generators [32] can automatically insert runtime checks, to improve the robustness. In our experience, generated code, thanks to its structural properties, is often automatically memory-safe by construction.

The costs and effort of using coding guidelines are moderate since no new programming language is involved. For an effective use, static code analysis tools need to be obtained and integrated into development processes, and developers need to learn how to interpret their results. Moreover, guidelines can restrict implementation choices and may require creating justifications and deviation records when rules need to be side-stepped or when analysis tools yield superfluous warnings ("False Positives"). Furthermore, coding guidelines have an indirect impact on runtime overhead, as they may favor constructs requiring more memory or processing time. In C++, for example, `std::vector::at()` includes implicit runtime boundary checks, unlike `std::vector::operator[]`. Similarly, defensive programming leads to more runtime checks explicitly added by the developer.



Status: Unreviewed
Severity: Unset
Enter comment here...

✓ **Out of bounds array index**
Array index is within bounds : [0..9]
array size: 10
array index value: [0 .. 9]

```
void initialise_current_data(void) {
    int i;

    current_data = &first_payload;
    for (i = 0; i < MAX_SIZE; i++) {
        tab[i] = 12;
    }
}


void partial
```

Assignment to element of global array (int 32): 12

array size: 10
array index value: [0 .. 9]

Press 'F2' for focus

Figure 4: Proof of Memory Safety in C++ with Formal Methods [34].

Limitations to applicability: Not all languages have widely accepted coding guidelines, which can be problematic in the context of certification. Retroactively applying guidelines to legacy code involves a high code churn, which is costly and may introduce new errors. Performance-sensitive applications may be adversely affected by the strict rules.

C. *Formal Verification Methods*

Formal Methods offer exhaustive detection of certain error classes, enabling the total elimination of memory errors and their impacts. Standards such as ISO 26262 [14] and IEC 61805 [15] recommend them, and numerous institutions like CISA [8] and NIST [33] highlight their drastic impact on error reduction. Through mathematical analysis, all possible control and data flows are considered, and the safety of each program operation is either disproven or proven. This is a significant difference from "normal" static code analysis tools, which only provide indications of errors but cannot confirm their absence [3].

In practice, Formal Methods are used similarly to static code analysis: An automated code review is conducted by parsing and analyzing the source code. Errors are reported to developers either directly within the development environment or via a CI system. Typical tools are based on Abstract Interpretation or Theorem Proving and widely used in the embedded domain (e.g., [34] for C/C++). An example of a safety proof is shown in Figure 4 shows an example of a safety proof, where Formal Verification could prove that an array access in C++ is safe, regardless of input parameters.

Formal Methods can reduce the need for runtime checks and help reclaiming performance: If they can prove that an error cannot occur under any condition, then defensive coding and explicit runtime checks can be removed safely, reducing memory footprint and execution time. This is particularly useful when coding guidelines would be prohibitive for performance.

The costs of using Formal Methods are comparable to those of coding guidelines. However, if applied naively, they can generate numerous false positives [1], and thus increase the review workload. Advanced usage tips have been published in [35].

Limitations to applicability: Formal Methods are a white-box approach and require access to the source code or modeling language. They may suffer from scalability problems when software is poorly structured and lacks interface definitions, in particular when used on complex software. Lastly, there is sometimes a lingering resistance against using Formal Methods, since early methods had a steep learning curve.

D. *Compiler Extensions*

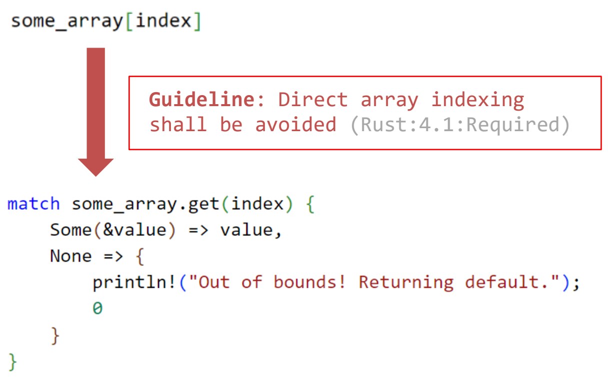
Compiler extensions offer limited reduction of underlying errors but aim to mitigate the impact of errors. While stricter compilation checks are possible through compiler plugins [36], they are practically constrained by the expectation for fast compilation times. Therefore, typical compiler extensions insert additional runtime checks into the instruction stream, which can be applied to both data and control flows and are sometimes performed by default according to language semantics (e.g., Ada).

They are also available for "memory-unsafe" languages like C and C++, for example:

- Clang Sanitizers [36] can detect invalid memory access like use-after free and out-of-bounds access.
- Control Flow Guard [37] monitors the integrity of control flow by validating the target address of indirect branches.
- CastGuard [8] partially detects type confusion, by detecting illegal downcasts.
- GWP-Asan [38] offers partial detection of heap memory errors related to allocation.

All these run-time measures, however, create a performance penalty as the processor executes additional check instructions. This can range from a few percent [8] [37] to several times the execution time [36] [39], depending on the benchmark and the use of language-specific features.

Another approach is taking measures to reduce error impacts, instead of reducing the errors themselves. This comes as a lower cost but does not help detecting logical errors. One such example is automatic initialization of heap or stack variables [8].



```
some_array[index]
```

Guideline: Direct array indexing shall be avoided (Rust:4.1:Required)

```
match some_array.get(index) {
    Some(&value) => value,
    None => {
        println!("Out of bounds! Returning default.");
        0
    }
}
```

Figure 5: Coding guidelines help reducing run-time errors, also in memory-safe languages (here: Rust). The upper construct can result in a run-time panic and terminate the program, the lower is robust for all inputs.

Limitations to applicability: Certification of custom extensions are difficult to qualify for safety-critical applications, and the performance overhead can be prohibitive.

E. Memory-Safe Hardware

Memory-safe hardware aims solely to minimize the impact of memory errors. This approach relies on runtime checks like compiler extensions but implemented in hardware to keep overhead low. Memory errors are not reduced, but rather detected, with their impact minimized by stopping the program.

A common mechanism is the Memory Management Unit (MMU) used with an operating system, triggering a segmentation fault when accessing an invalid memory area. However, this method can only detect a subset of memory errors. More advanced methods, like memory tagging extensions to prevent heap corruption [3] [8], are already available but do not offer complete detection.

Another hardware influence on memory safety is the choice of architecture. Unlike the widely used von Neumann architecture, the Harvard architecture reduces the likelihood of control flow corruption by separating data and instruction memory.

Limitations to applicability: Availability and cost of the new hardware are limiting factors. For legacy systems, new hardware is typically prohibitive regarding cost and risk.

F. Safe(r) Libraries

Safer libraries can offer a moderate reduction in error causes and a minor reduction in the impact of errors with comparably low effort. Standard libraries can be replaced with safer variants that include more implicit runtime checks and stricter APIs. This can prevent some errors and detect others. An example is the use of safer C++ libraries with enhanced bounds-checking in the Chromium project [11] and in the Google codebase [13]. The C++ working group also supports this idea [17] and plans extensions in upcoming releases.

One specific example is Google's "Miracle Pointer" [40], which protects against use-after-free (i.e., temporal) memory errors. The main cost is memory overhead, typically around 5%.

Limitations to applicability: The availability of safe libraries is limited, and there is a lack of standardization, making certification and verification more challenging.

G. Dynamic Tests and Fuzzing

Testing is a widely used method to detect programming errors, and thus to reduce their prevalence. It is mandatory in many regulatory standards [15] [14] [17]. It involves executing the software with specific test cases and passing well-defined checkpoints. Fuzz testing [1] is particularly effective at detecting memory errors [13] and often relies on genetic algorithms [41] to find offensive inputs leading to memory errors. In all cases, the key to effective testing is having enough well-chosen test cases. While these can be automatically generated to achieve higher coverage (e.g., [42] for C/C++), they usually do not cover all program states. Therefore, dynamic tests can only detect the *presence* of errors, but not their *absence* [3] [33]. Some tools (e.g., [42]) try to overcome this limitation by integrating dynamic test with static analysis and Formal Methods.

Testing typically incurs high costs [43]. Additionally, it often introduces significant runtime overhead and hence cannot be used in the production environment. Tests are sometimes implicitly executed and evaluated in a "virtual machine" (e.g., Valgrind [44] or Miri [45]) to detect more error classes.

Limitations to applicability: The developer and tester should be independent to avoid testing bias. This can be prohibitive in terms of human and financial resources. Testing is difficult on incomplete code and may be infeasible (without further measures) for software under real-time constraints. Legacy systems may lack interfaces for effective testing.

IV. A Robust Development Process for Memory Safety

No single method can provide (more) memory safety without any additional effort or limitations, and more importantly, none of the methods alone can guarantee freedom from memory-related errors in critical software. Only an appropriate combination of methods and a well-defined development process are sufficient to produce robust software.

This can be seen by taking a closer look at the CrowdStrike incident, which would likely not have been prevented by choosing a different programming language or by applying coding guidelines—the software in question was a Windows device diver written in C++, a language considered memory-unsafe. The underlying logical error has been a mismatch between two array lengths, which ultimately led to an out-of-bounds memory read [46]. One of the arrays was instantiated from dynamic input data delivered via network as a "Channel File". This file was consumed by a content interpreter in the driver, and subsequently matched against a second array containing live sensor data. Depending on the system and its live data, the matching process could reach a state where only 20 inputs were provided, but the interpreter was trying to access the 21st one.

Regardless of the programming language, this discrepancy between dynamic input data and program structure cannot be detected by a compiler, since contextual information (from the channel file) is missing during compilation time. On the other hand, the compiler also cannot simply reject the program on grounds of potential errors, since the input may also be valid. Thus, the compiler will not consider the code as unsafe, and the compilation will be successful.

At run-time, the choice of programming language would not have made much difference, either: The driver's memory access error was detected by the Windows kernel. At this point, the kernel could not deactivate the driver, since this requires the cooperation of the driver itself, which was obviously in a bad state. Consequently, the kernel was forced to throw a Blue Screen to protect the integrity of the user data. A memory-safe language could have detected the error via run-time checks before the kernel. The result would have been the same since exception handling is not allowed in kernel contexts. Considering this information, it is questionable if the incident could have been avoided by using a memory safe programming language like Rust.

Dynamic testing has been used and presumably discovered bugs before deployment, but apparently all automated test cases merely covered a regex wildcard criterion in the 21st field. Thus, no test was able to expose the out-of-bounds read when provided 20 inputs instead of 21 [46]. Detecting software (error) states not covered by any test case is not trivial, even if the strictest coverage metrics like MC/DC are used. Therefore, testing can only be used to show to presence of bugs, not their absence. Fuzz Testing could have increased the chances of finding the bug. However, it could not provide any guarantee of absence either since it typically struggles to reach a high coverage [41].

Coding guidelines, in principle, may have suggested the use of safer containers, explicit error handling, and bounds-checked access, increasing risk awareness. However, they could also create non-negligible performance penalties, a concerning factor in kernel drivers. In such a context, they may not produce additional value. A guideline violation is not necessarily a bug—developers may prioritize performance over style. Hence, coding guidelines may not be a good choice to increase memory safety in a kernel driver.

Using Formal Methods would have reported clear evidence of all error cases and—according to our replication study—identified this bug. However, it does not solve the problem on its own but requires additional process steps. The analysis would have presented the issue as a *potential* bug that depends on input conditions and pointed to many other potential problems. It is then the task of the developer to review the findings, provide external context (like the contents of the channel file) to refine the analysis, and—as last measure—add defensive code to prove the safety of the array access operation.

Other process measures could have been a staggered deployment to minimize risk (e.g., non-critical clients first [46]) and architectural changes in the driver that move more functionality into the userland to reduce the probability of a Blue Screen (however, feasibility depends on the OS and its interfaces).

In summary, we can see that the benefits of each method depend on the type and the context of the software being developed. Hence, we must carefully evaluate each method in the specific domain and context and then define the development and deployment process around them. There is no generic optimal solution to avoid memory-related errors. Only by selecting several methods, knowing their limitations, and using them consistently, we can keep the cost low and provide an effective solution to memory safety.

V. Summary and Recommendations

Memory safety is sometimes misunderstood, and there is no universal solution to completely eliminate the causes and effects of memory errors. Different approaches to memory safety are available, each aiming to either fundamentally prevent the underlying causes of memory errors or to mitigate their effects to varying degrees. Do not rely solely on the "memory safe" attribute—while it is a useful feature for embedded systems, it is insufficient on its own. Instead, we must consider how it is achieved, at what cost, and to what extent.

An effective memory safety strategy requires an appropriate combination of multiple methods to reduce or even eliminate both the occurrences and impact of underlying errors. Memory-safe programming languages offer clear advantages but must be combined with other methods, such as coding guidelines to guide developers towards safer patterns and Formal Methods to foresee and completely eliminate the negative effects of memory errors.

For existing software, an economically viable strategy might involve first employing safer libraries and better verification tools to identify existing errors. The next step is implementing new components in memory-safe languages and using hardware with memory-safe features. Finally, the re-development of legacy components can be considered to take advantage of new languages. Static code analysis and coding guidelines are essential at all stages to sustainably avoid runtime errors, regardless of programming language and hardware use. Through these incremental improvements, developers can enhance memory safety in embedded systems while addressing the challenges associated with integrating new technologies.

VI. References

[1] CISA, NSA, FBI, ASC, CCS, NCSC-UK, NCSC-NZ, CERT-NZ, "The Case for Memory Safe Roadmaps," 2023.

[2] DoD DARPA, "Translating All C TO Rust (TRACTOR)," 2024. [Online]. Available: https://sam.gov/opp/1e45d648886b4e9ca91890285af77eb7/view.

[3] The White House, "Back to the Building Blocks: A Path toward Secure and Measurable Software," Washington, USA, 2024.

[4] V. van der Veen, N. dutt-Sharma, L. Cavallaro and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Research in Attacks, Intrusions, and Defenses*, Amsterdam, Netherlands, 2012.

[5] Wikipedia, "Morris worm," [Online]. Available: https://en.wikipedia.org/wiki/Morris_worm.

[6] Wikipedia, "2024 CrowdStrike-related IT outages," [Online]. Available: https://en.wikipedia.org/w/index.php?title=2024_CrowdStrike-related_IT_outages&oldid=1249344762.

[7] Wikipedia, "Heartbleed," [Online]. Available: https://en.wikipedia.org/wiki/Heartbleed.

[8] CSAC Technical Advisory Council, "Report to the CISA Director, Memory Safety," 2023.

[9] Microsoft Research, "We need a safer systems programming language," [Online]. Available: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/. [Accessed 04 Oct 2024].

[10] Vander-Stoep, "Eliminating Memory Safety Vulnerabilities at the Source," Google Security Blog, 2024. [Online]. Available: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html.

[11] Chromium Project, "Memory safety," 2020. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/.

[12] Google, Inc., "Project Zero," 2022. [Online]. Available: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[13] A. Rebert, C. Carruth, J. Engel and A. Qin, "Safer with Google: Advancing Memory Safety," Google, 2024. [Online]. Available: https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html. [Accessed 16 12 2024].

[14] International Organization for Standardization (ISO), "International Standard ISO 26262-6 (Road Vehicles - Functional Safety)," Geneva, Switzerland, 2018.

[15] IEC, "61508-3, Functional Safety, Software Requirements," Geneva, Switzerland, 2010.

[16] J. Beningo, "Has C++ Just Become More Popular than C?," 2024. [Online]. Available: https://www.embedded.com/has-c-just-become-more-popular-than-c/.

[17] H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde and M. Wong, "DG OPINION ON SAFETY FOR ISO C++," WG21, 2023.

[18] P. Whytock, "From C++ to Rust: The Changing Landscape of Automotive Programming," 2023. [Online]. Available: https://www.electropages.com/blog/2023/06/the-changing-face-of-automotive-aoftware.

[19] J. Foufas, "Why Rust is actually good for your car," 2022. [Online]. Available: https://medium.com/volvo-cars-engineering/why-volvo-thinks-you-should-have-rust-in-your-car-4320bd639e09. [Accessed 04 Oct 2024].

[20] "Rust in Automotive," reddit, [Online]. Available: https://www.reddit.com/r/rust/comments/yuwoby/rust_in_automotive/. [Accessed 04 Oct 2024].

[21] M. Becker and J. Palczynski, "Automatic Verification of (un)intended Data and Control Flows in Embedded Software," in *Embedded World Conference*, Nuremberg, Germany, 2024.

[22] CISA, "BadAlloc Vulnerability Affecting BlackBerry QNX RTOS," 23 Aug 2021. [Online]. Available: https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-229a.

[23] Wikipedia, "Therac-25," [Online]. Available: https://en.wikipedia.org/w/index.php?title=Therac-25&oldid=1254594365.

[24] NSA, "Software Memory Safety," NSA, 2023.

[25] AdaIC, "The Boeing 777 Flies 99.9% on Ada," 2010. [Online]. Available: https://web.archive.org/web/20101229070248/http://archive.adaic.com/projects/atwork/boeing.html. [Accessed 04 Oct 2024].

[26] S. Klabnik, "Memory Safety is a Red Herring," 2023. [Online]. Available: https://steveklabnik.com/writing/memory-safety-is-a-red-herring/. [Accessed 04 Oct 2024].

[27] S. Baxter and C. Mazakas, "Safe C++," 2024. [Online]. Available: https://safecpp.org/P3390R0.html.

[28] A. Alheraki, "Will C++26 Solve the Memory Safety Issue?," 2024. [Online]. Available: https://simplifycpp.org/?id=a0310.

[29] MISRA Consortium, "MISRA C++:2023 Guidelines for the use C++:17 in critical systems," 2023.

[30] MathWorks, Inc., "Polyspace Bug Finder," [Online]. Available: https://mathworks.com/products/polyspace-bug-finder.html.

[31] B. Qin, "A curated list of awesome Rust checkers," 13 July 2024. [Online]. Available: https://burtonqin.github.io/posts/2024/07/rustcheckers/. [Accessed 16 Dec 2024].

[32] MathWorks, Inc., "Embedded Coder," [Online]. Available: https://uk.mathworks.com/products/embedded-coder.html. [Accessed 2024].

[33] DoC NIST, "Dramatically Reducing Software Vulnerabilities," 2016.

[34] MathWorks, Inc., "Polyspace Code Prover," 18 Oct 2021. [Online]. Available: https://www.mathworks.com/products/polyspace-code-prover.html.

[35] M. Becker and J. Palczynski, "Increasing Resilience to Cyberattacks Through Advanced Use of Static Code Analysis," in *Embedded World Conference*, Nuremberg, Germany, 2021.

[36] LLVM, "Clang compiler user's manual, controlling code generation," [Online]. Available: https://clang.llvm.org/docs/UsersManual.html#controlling-code-generation. [Accessed 04 Oct 2024].

[37] Microsoft Research, "Control Flow Guard for Clang/LLVM and Rust," 2020. [Online]. Available: https://msrc.microsoft.com/blog/2020/08/control-flow-guard-for-clang-llvm-and-rust/.

[38] Chromium Project, "GWP-ASan," 2019. [Online]. Available: https://www.chromium.org/Home/chromium-security/articles/gwp-asan/.

[39] TechHara, "Performance — C++ vs Rust vs Go," [Online]. Available: https://medium.com/@techhara/performance-c-vs-rust-vs-go-a44cbd2cc882. [Accessed 04 Oct 2024].

[40] A. Taylor, B. Nowierski and K. Haro, "Use-after-freedom: MiraclePtr," Google, 2022. [Online]. Available: https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html. [Accessed 16 12 2024].

[41] Wikipedia, "AFL," [Online]. Available: https://en.wikipedia.org/w/index.php?title=American_Fuzzy_Lop_(software).

[42] MathWorks, Inc., "Polyspace Test," [Online]. Available: https://mathworks.com/products/polyspace-test.html.

[43] G. Myers, C. Sandler and T. Badgett, The Art of Software Testing, Wiley, 2011.

[44] J. Seward and N. Nethercote, "Valgrind," 2000. [Online]. Available: https://valgrind.org/. [Accessed 04 Oct 2024].

[45] Rust community, "rust-lang miri," [Online]. Available: https://github.com/rust-lang/miri. [Accessed 16 Dec 2024].

[46] CrowdStrike Holdings, Inc., "External Technical Root Cause Analysis — Channel File 291," 6 August 2024. [Online]. Available: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf. [Accessed 06 January 2025].

[47] Microsoft, "Safe Libraries: C++ Standard Library," 2021. [Online]. Available: https://learn.microsoft.com/en-us/cpp/standard-library/safe-libraries-cpp-standard-library.

[48] M. Becker, "Let's talk about Memory Safety," 2024. [Online]. Available: https://www.linkedin.com/pulse/lets-talk-memory-safety-martin-becker-tdw8f/. [Accessed 04 Oct 2024].

[49] Google Inc., "Google Trends "Memory Safety"," [Online]. Available: https://trends.google.com/trends/explore?date=2005-01-01%202024-12-13&q=%2Fm%2F03c2cg9&hl=en. [Accessed 13 Dec 2024].