AdaCore TECH PAPER

Memory Safety in Ada, SPARK, and Rust

Memory Safety in Ada, SPARK, and Rust

Executive Summary

Memory safety bugs pose a significant threat to the security and reliability of critical software systems, with major tech companies like Google and Microsoft attributing over 70% of their security vulnerabilities to these issues. Addressing memory safety at the programming language level is essential for mitigating these risks.

This whitepaper explores the most common memory safety bugs, including out-of-bounds writes, use after free errors, and null pointer dereferences, and highlights how specific programming languages—Ada, SPARK, and Rust—provide robust solutions to these problems.

Ada and SPARK, both strongly typed languages, offer comprehensive memory safety through a combination of language features, runtime checks, and static analysis tools. Ada ensures safe memory management with features like array bounds checking and safe pointers, while SPARK eliminates many memory safety risks through static analysis and formal verification.

Rust introduces an innovative approach to memory safety with its ownership model, borrowing rules, and strong compile-time guarantees. Rust's memory safety features, such as bounds checking and null pointer elimination, make it a compelling choice for systems requiring both security and performance.

Adopting memory-safe programming languages like Ada, SPARK, and Rust can significantly reduce the incidence of memory-related bugs, leading to more secure and reliable software systems. These languages enable developers to build complex systems with confidence, knowing they are protected from some of the most challenging and persistent vulnerabilities in software development.

Introduction

Over the past decades, memory safety bugs have been a persistent and severe threat to the security and safety of some of our most critical cyber systems. The programming language is a crucial factor: it can be part of the problem or part of the solution.

Google (in 2021) and Microsoft (in 2019) revealed that more than 70% of their security bugs are due to memory safety issues.

Memory safety, a key concept in software development, addresses how an application handles memory operations, such as reading, writing, allocation, and deallocation. A memory-safe application operates within the bounds of its allocated memory -- it doesn't access or modify memory locations that it's not allowed to access –, avoids data races, and frees memory after use. Improper management of memory could result in severe problems ranging from crashing the application to security vulnerabilities that attackers can exploit.

Numerous software bugs can lead an application to read or write outside its allocated memory space. To avoid bugs, safety measures should be enforced either manually or by using memory-safe programming languages. While manual application of these measures is tedious and error-prone, the latter option provides dynamic and/or static built-in checks and tools to support the aim with minimum manual effort. The second option is in line with the recommendation of the US National Security Agency.

The Most Common Memory Bugs

Out-of-bounds Write:

Out-of-bounds writes occur when a program writes data outside the allocated memory range, potentially corrupting adjacent memory.

According to the Common Weakness Enumeration 2023, this is the most dangerous and stubborn software weakness. The write might happen on sensitive data, such as data used by other subprograms or the return address of a function. Attackers can exploit overwriting a return address to control the execution flow of a program and execute their injected code.

The root cause of this bug might be issues such as uncontrolled pointer arithmetic or dangling pointers.

Use After Free: (also known as dereferencing dangling pointers)

"Use after free" bugs occur when a program attempts to access memory after it has been deallocated.

A pointer to a memory location that already has been freed is called a dangling pointer, and dereferencing it could have a variety of negative consequences. If the freed memory is reused already, dereferencing the dangling pointer might corrupt the valid data (i.e., by overwriting it), or if the new data is a function return address, it can be exploited to execute malicious code. Even only reading data in the reused memory can cause sensitive information leaks.

Out-of-bounds Read:

This occurs when an application reads data beyond the intended buffer, resulting in the program's unexpected behavior or security vulnerabilities. The primary security issue is accessing sensitive data. More sophisticated threats could be executing arbitrary code or crashing the application.

Null Pointer Dereference:

Dereferencing a null pointer can cause a program to crash or behave unpredictably.

When an application attempts to access data through a pointer that does not point to a valid memory address, it normally crashes if exception handling is not in place.

Integer Overflow or Wraparound:

When arithmetic operations on an integer result in a value that exceeds the boundary of that type, that value wraps around to its maximum or minimum, depending on the operation. It can cause buffer overflow, infinite loop (if the value is a loop index), or resource (e.g., memory or CPU) exhaustion when the result of the calculation is used for resource management.

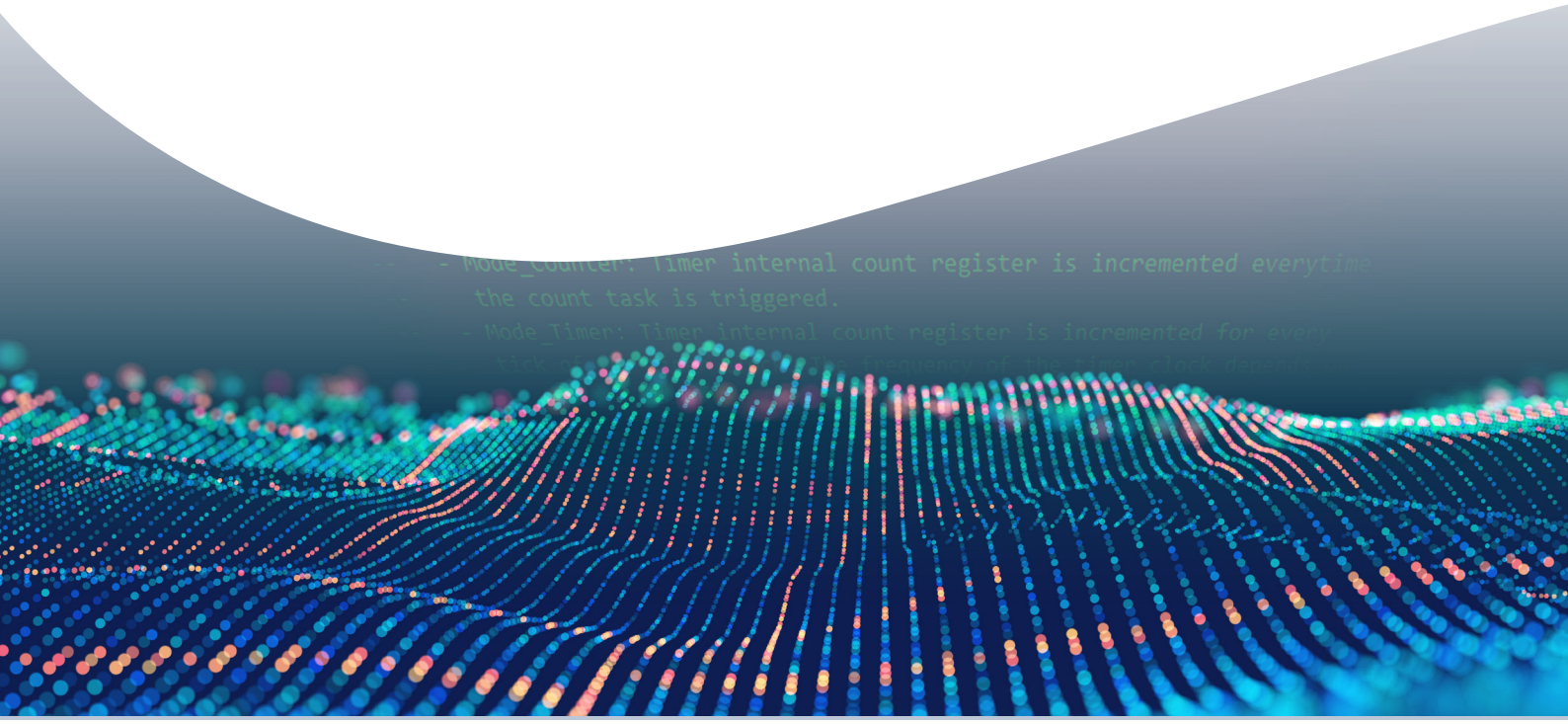
Data Race:

This refers to a situation where a resource that should be exclusively accessed is instead modified/read at the same time by concurrent threads. A race condition is due to the lack of synchronization between threads. The consequences could be unexpected states of the shared resource, application crashes, and resource exhaustion.

Memory leaks:

Memory leaks are a common class of memory-related bugs that occur when a computer program incorrectly manages memory allocations. Specifically, a memory leak happens when a program allocates memory but fails to release it back to the system after it is no longer needed. Over time, this can cause the application to consume increasing amounts of memory, leading to degraded performance and, in extreme cases, system crashes.

Memory leaks typically occur in languages that require manual memory management, such as C and C++. Programmers must explicitly allocate and deallocate memory, often leading to scenarios where memory is forgotten or mishandled. In contrast, languages like Ada and Rust incorporate safer memory management practices.



Memory-Safe Programming Languages

A memory-safe programming language enforces measures to prevent memory misuse instead of relying on developers to add proper checks in their code. These measures range from the most conventional, such as bounds checking, to more sophisticated ones, such as variable ownership.

Memory-safe languages help protect against these bugs and manage memory automatically rather than relying on programmer-provided checks, thus mitigating most memory safety issues. However, they differ in the level of protection, mechanisms, and support tools.

Ada and SPARK are strongly typed languages that provide memory safety through a combination of language features, run-time checks, and support for static and dynamic analysis tools.

Ada is a safe programming language widely used for developing safety-critical applications. It enforces memory safety through language constructs that entail extensive run-time and compile-time checks. Strong typing, formal parameter modes, protected objects, and safe pointers are among those constructs.

The built-in checks prevent errors, including buffer overflow and out-of-bounds read/write. Strong typing and parameter modes prevent injecting wrong data that modifies memory or possibly alters control flow. Protected objects ensure that accessing shared resources is safe and free of race conditions. Pointers, the leading cause of many memory bugs in other languages, are safer in Ada by provisioning safe accessibility rules and null exclusion, which prevents null pointer dereferencing. Furthermore, some features provide similar functionality to pointers but without the overhead and potential memory safety issues so that pointers do not need to be used.

Moreover, Ada's memory safety is backed by dynamic and static analysis tools that detect errors that are often hard to track down during runtime and compilation.

SPARK is a subset of Ada and is listed as a memory-safe language by NIST. It supports Ada's applicable memory safety features; a difference is that in SPARK, all checks are statically proven, whereas in Ada, many are enforced at run time. SPARK includes a safe pointer facility based on an ownership mechanism.

Rust supports memory safety through several mechanisms, including variables' immutability, ownership, borrowing, and bounds checking. In Rust, variables are immutable by default, which helps to protect against unintended changes. The Rust compiler manages memory using a system of ownership, which checks that each value in the program has a unique owner. When the owner goes out of scope, the memory is automatically released. Borrowing allows multiple references to have access to a variable, but only one mutable reference is possible at a time; this property is enforced in the presence of concurrency (threads), which prevents race conditions. Compile-time flow analysis checks that each pointer dereference is valid (i.e., the pointed-to value exists). There is no concept of a null pointer (thus, no way to dereference a null pointer). Run-time bounds checking verifies that the code accesses memory within the bounds of arrays and vectors, preventing buffer overflow.

Rust has a mode called "unsafe Rust,"' which allows programmers to disable some safety measures for more flexibility. However, unsafe Rust doesn't disable the borrow checker or bounds checking.

How Ada Addresses Common Memory Bugs

Ada is designed with robust safety features to mitigate common memory bugs. Below is an overview of how Ada handles each of these critical issues:

1. Out-of-Bounds Write

Ada addresses this issue through array bounds checking. During runtime, Ada automatically checks that any array access is within the valid range, raising a `Constraint_Error` exception if an out-of-bounds access is attempted. This prevents memory corruption and potential security vulnerabilities.

2. Use After Free

Ada's memory management features help to prevent this by avoiding manual memory management in most cases. For dynamic memory, Ada provides controlled types and storage pools that manage memory more safely. Additionally, Ada's strong typing and scope-based memory management reduce the chances of accidentally using freed memory.

3. Out-of-Bounds Read

Ada's runtime checks ensure that any array or string access is within its bounds, immediately raising an exception if a read operation is out of bounds. This prevents unintentional data leaks and undefined behavior.

Ada performs several run-time checks, and among them, the following contribute to mitigating memory safety errors. If a check fails, an exception is raised that can be handled properly to prevent unsafe conditions.

Index Check:

Ada performs bounds checking on array accesses and operations at run-time to verify that the index used to access a specific component is within the array bounds. This ensures that memory outside the bounds of an array is not accessed, preventing buffer overflows. Failing an Index Check will raise `Constraint_Error` and prevent undefined / unsafe behavior.

Excerpt showing Index Check fails.

```
procedure  Ex_Index_Check is
   type Integer_Array is
     array (Positive range <>) of Integer;
   function Inc_Value_Of (A : Integer_Array; I : Integer)
                     return Integer
   is
      begin
      return A (I) + 10;
   end Inc_Value_Of;
   Arr_1 : Integer_Array (1 .. 10) := (others => 1);
   Arr_2 : Integer_Array (1 .. 5) := (others => 0);
begin
   -- raises Constraint_Error as Index Check fails
   -- in the call of Inc_Value_Of (Arr_2, 10)
   Arr_1 (10) := Inc_Value_Of (Arr_2, 10);
end Ex_Index_Check;
```

Overflow Check:

This verifies that the value of a scalar object is within the specified range of its type; with Ada, there is no wraparound. While wrapping scalars might be an intended behavior in some circumstances, it can lead to software bugs including memory issues. One example is when the result is used to determine the offset or size in memory allocation. Failing this check raises `Constraint_Error`. If wrapping semantics is desired, the programmer can use a modular (unsigned) integer type.

Excerpt showing Overflow Check fails

```
procedure  Ex_Overflow_Check is
  A : Integer := Integer'Last;
begin
   -- raises Constraint_Error due to integer wraparound
   A := A + 1;
end  Ex_Overflow_Check;
```

Range Check:

Verifies that a scalar value is within a specific range. Ada allows user-defined types and subtypes with a specified range (an invariant on values of the type or subtype). This is useful when the type represents values such as size, length, frequency, speed, etc., which have application-specific ranges. Range checks ensure that assignment to a variable does not violate the type/subtype invariant; a violation would introduce undefined behavior, possibly including unsafe memory accesses.

Excerpt showing Range Check fails

```
procedure Ex_Range_Check is
   subtype Sub_Int is Integer range 1 .. 50;
   I : Sub_Int;
begin
   I := 52;  -- raises Constraint_Error
end Ex_Range_Check;
```

Storage Check:

Verifies that when new objects are created, either on the stack or in a Storage Pool, there is enough memory space, and if not, the `Storage_Error` exception is raised. This check can help prevent out-of-bounds read and write.

Discriminant Check:

Ensures that accessing fields in a variant record is safe. When a program tries to access a field that is not valid for a variant record, `Constraint_Error` is raised, preventing Reliance on Data/Memory Layout and Access of Resource Using Incompatible Type ('Type Confusion') errors.

Excerpt showing Discriminant Check fails

```
procedure Ex_Discriminant_Check is
   type Message_Type is (Number, Pointer);
      type Message_Buffer (MT : Message_Type) is record
   case MT is
    when Number =>
           Num : Integer;
    when Pointer =>
           Ptr : access Integer;
end case;
   end record;
    Number_MB : Message_Buffer (MT => Number);
   Var : Integer := 0;
begin
   Number_MB.Num := 21; -- OK
   Var := Number_MB.Ptr.all;  -- raises Constraint_Error
end Ex_Discriminant_Check;
```

Length Check:

Makes sure that in array assignments both arrays have the same length. This helps to avoid out-of-bounds read and write in array operations.

Excerpt showing Length Check fails

```
procedure Ex_Length_Check is
   type Integer_Array is
     array (Positive range <>) of Integer;

   Arr_1 : Integer_Array (1 .. 10);
   Arr_2 : Integer_Array (1 .. 9) :=
             (others => 1);
begin
  -- Length Check fails --> raises Constraint_Error
  Arr_2 := Arr1;
  end Ex_Length_Check;
```

Accessibility Check:

This check, introduced earlier, prevents dangling references for access values designating declared objects (the checks apply to access types designating subprograms as well, but that does not involve memory management). The check can be done at compile-time, generally, because access objects cannot exist longer than their corresponding types, and the compiler can use that fact to determine if the possibility for a dangling reference exists.

4. Null Pointer Dereference

Ada mitigates this risk by requiring explicit checks for null values when accessing pointers. Ada's `Access` types (pointers) have built-in nullity checks, and dereferencing a null access value will raise a `Constraint_Error` exception, making it less likely for null pointer dereferences to go unnoticed.

In other languages, pointers are the main cause of many memory bugs, such as dereferencing a null or out-of-bounds pointer, use after free, and double free.

Ada protects programmers from pointer dangers in the following ways:

Pointers in Ada are implemented through a more abstract, machine-independent facility known as "access types." Values of these types have a default initial value that designates no object and can designate both dynamically allocated objects as well as declared objects (i.e., those that are on a notional stack). Values designating allocated objects are created via the reserved word "new," as in some other languages. Values designating declared objects are created via the attribute 'Access, applied to the declared object itself. No other means of creating values exists, absent an explicitly unchecked conversion, so the facility is safer than those that allow arbitrary creation of pointer values, say via addresses. The only way to deallocate is via an explicitly unchecked facility. Thus, an access value is always meaningful, absent explicit use of an unchecked facility.

Providing alternative constructs that can be used instead of pointers. Parameter modes, unconstrained arrays, and dynamically sized objects without using a heap are constructs that can replace typical pointer usages.

5. Integer Overflow

Ada provides both compile-time and runtime checks for integer overflow. By default, Ada raises a `Constraint_Error` exception on overflow unless the developer explicitly disables checks with specific pragmas, ensuring that integer operations remain safe and predictable.

6. Data Race

Ada prevents data races through its tasking model, which is based on the concept of protected objects and rendezvous. Protected objects provide safe, concurrent access to shared data by enforcing mutual exclusion. Furthermore, Ada's strong typing and tasking constructs ensure that race conditions are less likely to occur, promoting safe concurrent programming.

7. Memory leaks

Ada is a language that minimizes the need for dynamic memory allocation. Its high-level, abstract types often eliminate the need for dynamically created objects. For instance, variables on the stack can have sizes that are not determined at compile time, unconstrained types can be returned without relying on dynamic memory, and collection types in Ada even handle memory management automatically.

The absence of dynamic allocation can be enforced programmatically. When dynamic memory is needed, Ada provides controlled types to automate cleanup as objects go out of scope, and user-defined storage pools enable the collective deallocation of dynamically allocated objects as needed.

By incorporating these safety features, Ada significantly reduces the risk of common memory bugs, contributing to the development of reliable and secure software systems.

How Rust Addresses Common Memory Bugs

Rust is designed with strong guarantees around memory safety, offering robust protection against common memory bugs. Here's how Rust handles each of these critical issues:

1. Out-of-Bounds Write

Rust's safety mechanisms, particularly its ownership system and the use of safe indexing, prevent this. When accessing arrays or slices, Rust automatically checks that the index is within bounds. If an out-of-bounds write is attempted, Rust will immediately panic, stopping execution and preventing memory corruption.

2. Use After Free

Rust's ownership and borrowing system ensures that memory is automatically and safely deallocated when it goes out of scope. The Rust compiler enforces strict rules around ownership, borrowing, and lifetimes, which effectively prevent use after free errors. Memory is only freed when it is no longer accessible, making it impossible to use freed memory.

3. Out-of-Bounds Read

Rust's built-in bounds checking for arrays and slices ensures that any read operation is within the valid memory range. If an out-of-bounds read is attempted, Rust will trigger a panic, preventing unintended data access and undefined behavior.

4. Null Pointer Dereference

Null pointer dereferencing occurs when a program tries to access a memory location through a pointer that is null, leading to crashes or unexpected behavior. Rust eliminates this risk by not allowing null pointers in safe Rust. Instead, Rust uses the `Option<T>` type to represent potentially absent values. This type forces the programmer to explicitly handle the case where a value might be absent, thereby preventing null pointer dereferences.

5. Integer Overflow

Rust offers two modes of handling integer overflow. In debug builds, Rust checks for overflow during arithmetic operations and will panic if an overflow occurs. In release builds, Rust does not perform these checks by default, but it offers checked, wrapping, and saturating arithmetic operations that allow developers to handle overflow explicitly, ensuring that integer operations are safe and predictable.

6. Data Race

A data race happens when two or more threads access shared data concurrently, and at least one of the accesses is a write without proper synchronization. Rust's ownership model and type system prevent data races at compile time. Rust enforces strict borrowing rules that prevent data from being accessed by multiple threads in an unsafe manner. Additionally, Rust provides concurrency primitives like `Mutex` and `RwLock` that ensure safe access to shared data. The compiler checks that these primitives are used correctly, thus eliminating the possibility of data races.

7. Memory leaks

Rust's ownership model enforces strict memory safety guarantees by ensuring that memory is automatically freed when it goes out of scope.

By enforcing strict rules at compile time and providing safe abstractions, Rust prevents many of the memory bugs that are common in other programming languages, making it an ideal choice for developing reliable and secure systems.

How SPARK Addresses Common Memory Bugs

SPARK, a formally verified subset of Ada, is designed for high-assurance systems where safety and security are paramount. SPARK eliminates many common memory bugs through its restrictive programming model, static analysis, and formal verification. Here's how SPARK addresses these critical memory issues:

1. Out-of-Bounds Write

SPARK completely eliminates this risk by performing static analysis at compile time. Through formal verification, SPARK ensures that all array accesses are within valid bounds. Any attempt to write outside of these bounds is detected before the program is executed, guaranteeing that such errors are impossible in verified SPARK code.

2. Use After Free

SPARK avoids this issue by not supporting dynamic memory allocation and deallocation in its safe subset. By disallowing operations that can lead to memory mismanagement, SPARK ensures that memory is allocated and used in a predictable and safe manner, effectively preventing use after free errors.

3. Out-of-Bounds Read

In SPARK, array bounds are rigorously checked at compile time using formal methods. This static analysis ensures that all reads are within the valid range of the data structure, eliminating the possibility of out-of-bounds reads in verified SPARK code.

4. Null Pointer Dereference

Null pointer dereferencing, which occurs when a program tries to access a memory location through a null pointer, is completely eliminated in SPARK. The language design forbids the use of null pointers, and instead, SPARK enforces that all pointer-like structures (known as `Access` types in Ada) are properly initialized and checked. This guarantees that dereferencing operations are always valid.

5. Integer Overflow

SPARK prevents this by performing static analysis to ensure that all integer operations are within safe bounds. The language requires that developers specify the expected ranges of variables, and SPARK's analysis tools verify that these ranges are respected throughout the program. This formal verification ensures that integer overflow cannot occur in SPARK-compliant code.

6. Data Race

SPARK eliminates the possibility of data races through its restrictive concurrency model. SPARK enforces strict rules around data access in concurrent programs, and the formal verification process ensures that these rules are adhered to. By disallowing unsynchronised access to shared data, SPARK prevents data races at the source.

7. Memory leaks

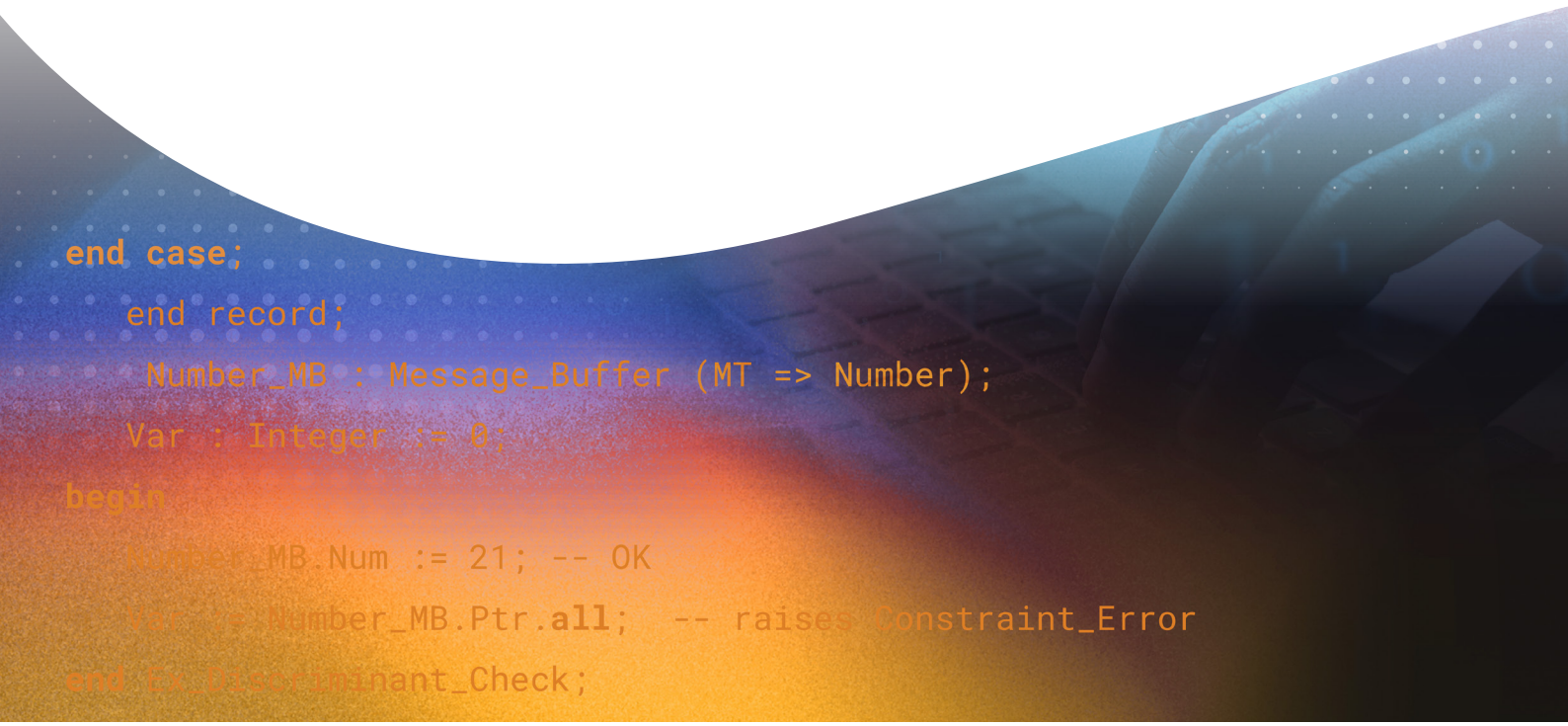
SPARK further enhances memory safety by eliminating certain programming constructs that can lead to memory misuse.

By leveraging formal verification and eliminating unsafe programming practices, SPARK ensures that the most common memory bugs are not just rare but impossible within verified code. This makes SPARK particularly suitable for applications where reliability and security are non-negotiable.

In conclusion, memory safety is essential to developing secure and reliable software systems, especially in environments where the consequences of memory-related bugs can be catastrophic.

Ada, SPARK, and Rust exemplify how modern programming languages can mitigate or even eliminate common memory safety issues through strong typing, compile-time checks, and runtime safeguards. Ada provides extensive runtime checks and safe language constructs, ensuring that memory is managed effectively and securely. SPARK, a formally verified subset of Ada, goes even further by using static analysis to eliminate memory safety risks before they can manifest, making it a prime choice for high-assurance systems. Rust's innovative ownership model, borrowing rules, and built-in safety mechanisms offer a balance of safety and performance, making it an ideal choice for systems that require both security and efficiency.

As memory safety continues to be a critical concern, adopting these languages can significantly reduce the prevalence of memory-related bugs, leading to more secure and dependable software. By integrating memory-safe practices at the language level, developers can focus on building complex systems with the confidence that they are protected against some of the most insidious and challenging vulnerabilities in software development.



AdaCore

adacore.com