Three Paths to Memory Safety For Embedded

building more secure embedded systems

Mark Hermeling

AdaCore | Build Software that Matters

www.embeddedonlineconference.com

AGENDA

1. Today's Reality
2. Memory Safety
3. Examples
4. Runtime Errors
5. Formal Methods and Ada SPARK
6. Agentic AI

Today's Reality

Embedded software continues to increase in size and complexity

The amount of safety critical software is increasing

Regulation around security is increasing

Today's Reality: Software teams are struggling

They need to increase productivity

They need to increase agility

With fewer resources

Today's Reality: The software problem

C and C++ are unsafe languages in how they handle memory

Most embedded systems are built using these languages

70% of security vulnerabilities are related to memory safety

Increasing Regulations

Safety and Security by Design

Regulations are making vendors responsible for the software they deliver.

They recommend vendors to use ‘software development best practices’.

Memory Safety

Definitions

Memory safety
The property that memory can not get corrupted due to buffer overruns, type conversions, uninitialized variables.
A memory safe language can still have runtime errors, but the program will halt, or throw an exception during runtime.

Absence of runtime errors
The property that the application cannot crash/halt unexpectedly. Does not guarantee that it does what it was designed to do.

Functional correctness
The property that the application behaves as expected. This requires proper description of what the intended behavior is.

Memory Safety Across Languages

C and C++	Rust	Ada / SPARK
• MISRA as a safer language subset • Static analysis to find deep run-time errors • Warning path display to accelerate remediation	• Rapidly evolving and popular language • Some Linux kernel driver usage • Mix of dynamic and static memory safety	• Proven-in-use memory safe & type-safe languages • Deployed in 1000s of projects • Ada has a mix of dynamic and static memory safety • SPARK has static memory safety

Examples

C Example

With great power comes great responsibility

- Buffer overrun line 38
- Silently ignored during compilation
- Ignored during run-time
- Caught by static analysis tool at compile time
- Caught by address sanitizer at runtime

```
    printf("\nBuffer overrun -- accessing lights[4] (one past end):\n");
    TrafficLight overrun = lights[4];          /* <-- UNDEFINED BEHAVIOUR */
```

Buffer Overrun

This code reads past the end of the buffer pointed to by `lights`.

- `lights` evaluates to `malloc(...)` buffer_overrun.c:25
- The first byte read is at offset 16 from the beginning of the buffer pointed to by `lights`, whose capacity is 16 bytes.
  - The offset exceeds the capacity.
- The overrun occurs in heap memory.

The issue can occur if the highlighted code executes.

See related events 3 and 4.
Show: All events | Only primary events

```c
typedef enum {
    OFF = 0,
    RED,
    YELLOW,
    GREEN
} TrafficLight;

static const char *light_name(TrafficLight t)
{
    switch (t) {
    case OFF:    return "OFF";
    case RED:    return "RED";
    case YELLOW: return "YELLOW";
    case GREEN:  return "GREEN";
    default:     return "UNKNOWN";
    }
}

int main(void)
{
    /* Dynamically allocate exactly 4 elements (valid indices: 0..3). */
    TrafficLight *lights = malloc(4 * sizeof(TrafficLight));
    if (!lights) { fputs("malloc failed\n", stderr); return 1; }

    lights[0] = OFF;
    lights[1] = RED;
    lights[2] = YELLOW;
    lights[3] = GREEN;

    printf("Valid accesses (indices 0-3):\n");
    for (int i = 0; i < 4; i++)
        printf("  lights[%d] = %s\n", i, light_name(lights[i]));

    printf("\nBuffer overrun -- accessing lights[4] (one past end):\n");
    TrafficLight overrun = lights[4];          /* <-- UNDEFINED BEHAVIOUR */
    printf("  lights[4] = %s (raw value: %d)  <-- garbage / UB\n",
           light_name(overrun), (int)overrun);

    free(lights);
    return 0;
}
```

Rust Example

Not all checks are static

- Buffer overrun on line 27
- Ignored during compilation
- Caught during runtime

thread 'main' (12875103) panicked at src/main.rs:27:26:
index out of bounds: the len is 4 but the index is 4

```rust
#[derive(Debug, Clone, Copy)]
enum TrafficLight {
    Off,
    Red,
    Yellow,
    Green,
}

fn main() {
    // Dynamically allocated Vec with exactly 4 elements (valid indices: 0..3).
    let lights: Vec<TrafficLight> = vec![
        TrafficLight::Off,
        TrafficLight::Red,
        TrafficLight::Yellow,
        TrafficLight::Green,
    ];

    println!("Valid accesses (indices 0-3):");
    for (i, light) in lights.iter().enumerate() {
        println!("  lights[{}] = {:?}", i, light);
    }

    // Buffer overrun attempt: index 4 is one past the end.
    // Safe Rust catches this at runtime and panics -- no undefined behaviour,
    // no silent memory corruption.  The process terminates with a clear message.
    println!("\nBuffer overrun -- accessing lights[4] (one past end):");
    let _overrun = lights[4]; // <-- runtime panic: index out of bounds
    println!("  (this line is never reached)");
}
```

Ada / SPARK Example

Absence of RunTime Errors

- Typically, would not compile
- You would have a type that limits the index
  - See lines 40-43
- Ada gives a run-time error
  - Using Overrun_Index
- SPARK gives a compile-time error

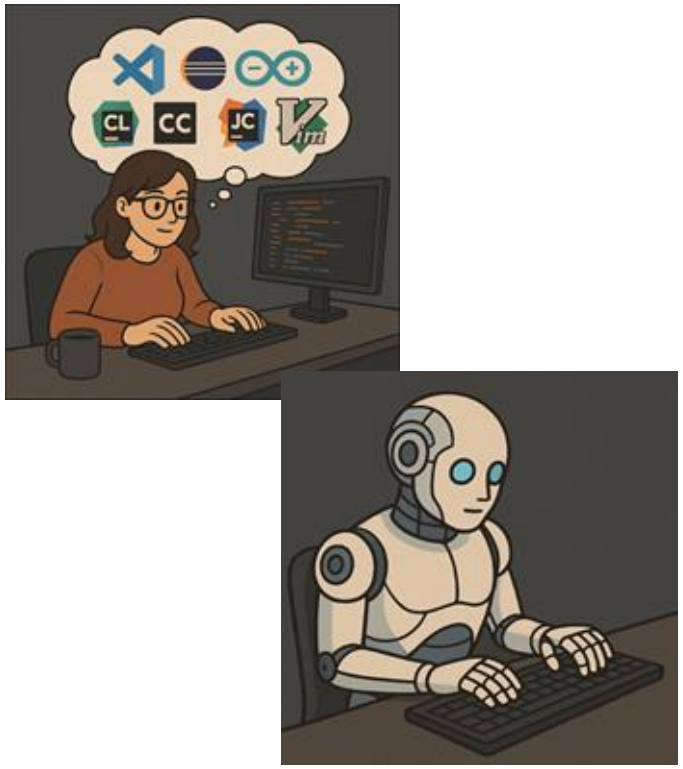
```
hermeling@bock buffer_overrun % make spark-prove
cd ada && alr exec -- gnatprove -P buffer_overrun.gpr --level=2
warning: attribute "Switches" of package "Prove" of your project file is deprecated
warning: use "Proof_Switches ("Ada")" instead
Phase 1 of 3: generation of data representation information ...
Phase 2 of 3: generation of Global contracts ...
Phase 3 of 3: flow analysis and proof ...

buffer_overrun.adb:51:58: high: range check might fail, cannot prove upper bound for Overrun_Index
   51 |            & Traffic_Light'Image (Lights (Valid_Index (Overrun_Index))));
      |                                                        ^~~~~~~~~~~~~
  e.g. when Overrun_Index = 4
  reason for check: value must be convertible to the target type of the conversion

buffer_overrun.adb:56:07: warning: this statement is never reached
   56 |      Put_Line ("  Caught Constraint_Error: 4 is outside Valid_Index (0 .. 3).");
      |      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

buffer_overrun.adb:57:07: warning: this statement is never reached
   57 |      Put_Line ("  Ada/SPARK prevented the buffer overrun.");
      |      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Summary logged in /Users/hermeling/eoc/buffer_overrun/ada/obj/gnatprove/gnatprove.out

hermeling@bock buffer_overrun %
```

```
27
28  procedure Buffer_Overrun is
29
30     type Traffic_Light is (Off, Red, Yellow, Green);
31     subtype Valid_Index is Integer range 0 .. 3;
32     type    Light_Array is array (Valid_Index) of Traffic_Light;
33
34     Lights        : constant Light_Array :=
35       (0 => Off, 1 => Red, 2 => Yellow, 3 => Green);
36     Overrun_Index : Integer := 4;  --  variable: check deferred to runtime
37
38  begin
39     Put_Line ("Valid accesses (indices 0 .. 3):");
40     for I in Valid_Index loop
41        Put_Line ("  lights(" & Integer'Image (I) & " ) = "
42                  & Traffic_Light'Image (Lights (I)));
43     end loop;
44
45     Put_Line ("");
46     Put_Line ("Buffer overrun -- accessing lights(4) (one past end):");
47
48     --  Valid_Index (Overrun_Index) raises Constraint_Error at runtime.
49     --  gnatprove flags this line statically: "high: range check might fail".
50     Put_Line ("  lights(4) = "
51               & Traffic_Light'Image (Lights (Valid_Index (Overrun_Index))));
52     Put_Line ("  (this line is never reached)");
53
54  exception
55     when Constraint_Error =>
56        Put_Line ("  Caught Constraint_Error: 4 is outside Valid_Index (0 .. 3).");
57        Put_Line ("  Ada/SPARK prevented the buffer overrun.");
58
59  end Buffer_Overrun;
```

Preventing Runtime Errors

Preventing Errors in C and C++

Developers Needs Help!



Approve Requires 1 approval from Critical Security Findings. Assign reviewers

Security scanning detected 3 new potential vulnerabilities
0 critical, 2 high and 1 others
View all pipeline findings

SAST detected 3 new potential vulnerabilities
0 critical, 2 high and 1 others

New
- High CWE-476 in terminal.c
- High CWE-120 in terminal.c
- Low CWE-482 in terminal.c

Null Pointer Dereference at terminal.c:46
Jump to warning location ↓
"Part of the SDK, not out concern..." | edit properties
warning details...

Show Events | Options

return_append_str() */builds/NgWdgbhnN/1/codesecure1/codesecure-se/demos/curl/src/terminal.c*

```
33  char *return_append_str(char *dest, const char *s) {
34  /* Append text s to dest, and return new result. */
35          char *new_loc;
36          size_t new_len;
37          /* This doesn't have buffer overflow vulnerabilities, because
38             we always allocate for enough space before appending. */
39          if (!dest) {
40                  new_loc = (char *) malloc(strlen(s))+1;
41                  strcpy(new_loc, s);
42                  return new_loc;
43          }
44          new_len = strlen(dest) + strlen(s) + 1;
45          new_loc = (char *) malloc(new_len);
```

Event 2: `malloc()` returns NULL.
- Dereferenced later, causing the null pointer dereference.

▲ ▼ hide

```
46          strcpy(new_loc, dest);
```

Null Pointer Dereference
The body of `strcpy()` dereferences `new_loc`, but it is NULL.

The issue can occur if the highlighted code executes.

See related event 4.
Show: All events | Only primary events

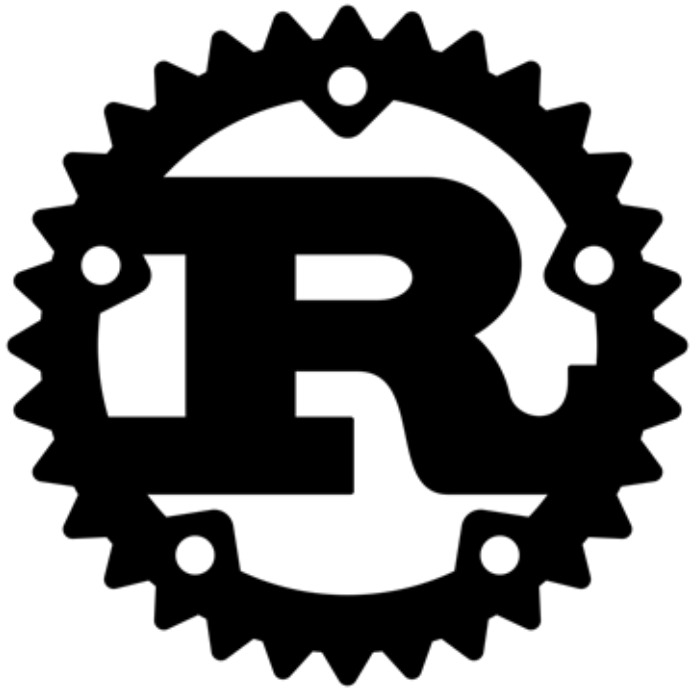
Preventing Errors in Rust

The Compiler Helps

Help through language features

- Static ownership and borrowing system
- Runtime elimination of null pointers using Option<T>
- Runtime bounds checking for arrays and slices

Still requires testing to find all dynamic errors
Still requires ‘unsafe’ code to perform device access



Preventing Errors in Ada / SPARK

Safe and Secure by Design

The language lifts the quality of the software
- Runtime checking in Ada

SPARK limits language features to make formal methods possible
- No exception handling
- No access types with allocations (pointers), borrow checker semantics allowed
- Limited tasking
- And adds contracts to drive **formal methods**

Formal Methods!

Proof that a program satisfies a specification

- For all inputs and all possible executions

How?
- Translate code into logical formulas – verification conditions
- Proof them through logic solvers (SMT solvers)

Which languages?
- Ada SPARK
- Work on C (Frama-C is well known)
- Work on Rust

Different Modes of SPARK

Enables incremental adoption

- Stone
  - Program sticks to the SPARK subset
- Bronze
  - Variables are initialized and used, data flow validation
- Silver
  - Proof of the absence of runtime errors
- Gold
  - Basic functional correctness, introduce post and invariants
- Platinum
  - Deeper functional correctness, more properties

Ada / SPARK

Let’s explore with an example how formal methods help deliver better software

binary_search.adb 8

src > binary_search.adb > { } Binary_Search > Search

```
1  pragma SPARK_Mode (On);
2
3  package body Binary_Search with SPARK_Mode is
4
   ✓Prove
5     procedure Search
6       (A      : in Array_Type;
7        Target : in Integer;
8        Result : out Search_Result)
9     is
10       Low  : Index_Type;
11       High : Index_Type;
12       Mid  : Index_Type;
13    begin
14       Result.Found := False;
15       Result.Index := 1;
16
17
18
19       -- Binary search loop with loop invariants
20       while Low <= High loop
21
22
23          -- Calculate middle index avoiding overflow
24          Mid := Low + (High - Low) / 2;
25
26          if A (Mid) = Target then
27             Result.Found := True;
28             Result.Index := Mid;
29             return;
30          elsif A (Mid) < Target then
31             Low := Mid + 1;
32          elsif A (Mid) > Target then
33             High := Mid - 1;
34          end if;
35       end loop;
36
37    end Search;
38
39 end Binary_Search;
```

binary_search.ads

src > binary_search.ads > { } Binary_Search > Search

```
1  pragma SPARK_Mode (On);
2
3  package Binary_Search with SPARK_Mode is
4
5     type Index_Type is range 1 .. 1000;
6     type Array_Type is array (Index_Type range <>) of Integer;
7
8     -- Result record to return both found status and index
9     type Search_Result is record
10       Found : Boolean;
11       Index : Index_Type;
12    end record;
13
14    -- Ghost function to specify that an array is sorted
15 --    function Sorted (A : Array_Type) return Boolean is
16 --       (for all I in A'Range => (for all J in I .. A'Last => A (I) <= A (J)
17 --    with Ghost;
18
19
20    -- Binary search procedure with formal contracts
   ✓Prove
21    procedure Search
22       (A      : in Array_Type;
23        Target : in Integer;
24        Result : out Search_Result);      (Alt+C) Duo Quick Chat
25 --    with
26 --       Pre  => Sorted (A),
27 --       Post => (if Result.Found then
28 --                   Result.Index >= A'First and then
29 --                   Result.Index <= A'Last and then
30 --                   A (Result.Index) = Target
31 --                else
32 --                   (for all I in A'Range => A (I) /= Target));
33
34 end Binary_Search;
```

6 SARIF Results | binary_search.gpr

LOCATIONS 2 | RULES 1 | LOGS 1 | Filter r

Line ↓	Message
binary_search.adb 5	
12	initialization of "Mid" proved
20	"Low" might not be initialized
20	"High" might not be initialized
24	"Low" might not be initialized
24	"High" might not be initialized
binary_search.ads 1	
24	initialization of "Result" proved

Mode: Bronze
Initialization Errors

binary_search.adb 9

src > binary_search.adb > { } Binary_Search > Search

```
pragma SPARK_Mode (On);

package body Binary_Search with SPARK_Mode is

   ✓Prove
   procedure Search
     (A      : in Array_Type;
      Target : in Integer;
      Result : out Search_Result)
   is
      Low  : Index_Type;
      High : Index_Type;
      Mid  : Index_Type;
   begin
      Result.Found := False;
      Result.Index := 1;
      Low  := A'First;
      High := A'Last;

      -- Binary search loop with loop invariants
      while Low <= High loop

         -- Calculate middle index avoiding overflow
         Mid := Low + (High - Low) / 2;

         if A (Mid) = Target then      (Alt+C) Duo Quic
            Result.Found := True;
            Result.Index := Mid;
            return;
         elsif A (Mid) < Target then
            Low := Mid + 1;
         elsif A (Mid) > Target then
            High := Mid - 1;
         end if;
      end loop;

   end Search;

end Binary_Search;
```

binary_search.ads

src > binary_search.ads > { } Binary_Search > Search

```
pragma SPARK_Mode (On);

package Binary_Search with SPARK_Mode is

   type Index_Type is range 1 .. 1000;
   type Array_Type is array (Index_Type range <>) of Integer;

   -- Result record to return both found status and index
   type Search_Result is record
      Found : Boolean;
      Index : Index_Type;
   end record;

   -- Ghost function to specify that an array is sorted
--    function Sorted (A : Array_Type) return Boolean is
--      (for all I in A'Range => (for all J in I .. A'Last => A (I) <= A (J)
--    with Ghost;


   -- Binary search procedure with formal contracts
   ✓Prove
   procedure Search
     (A      : in Array_Type;
      Target : in Integer;
      Result : out Search_Result);      (Alt+C) Duo Quick Chat
--    with
--      Pre  => Sorted (A),
--      Post => (if Result.Found then
--                  Result.Index >= A'First and then
--                  Result.Index <= A'Last and then
--                  A (Result.Index) = Target
--               else
--                  (for all I in A'Range => A (I) /= Target));

end Binary_Search;
```

Fixes, then go to Silver
Now we have Buffer and Type Overruns

13 SARIF Results | binary_search.gpr

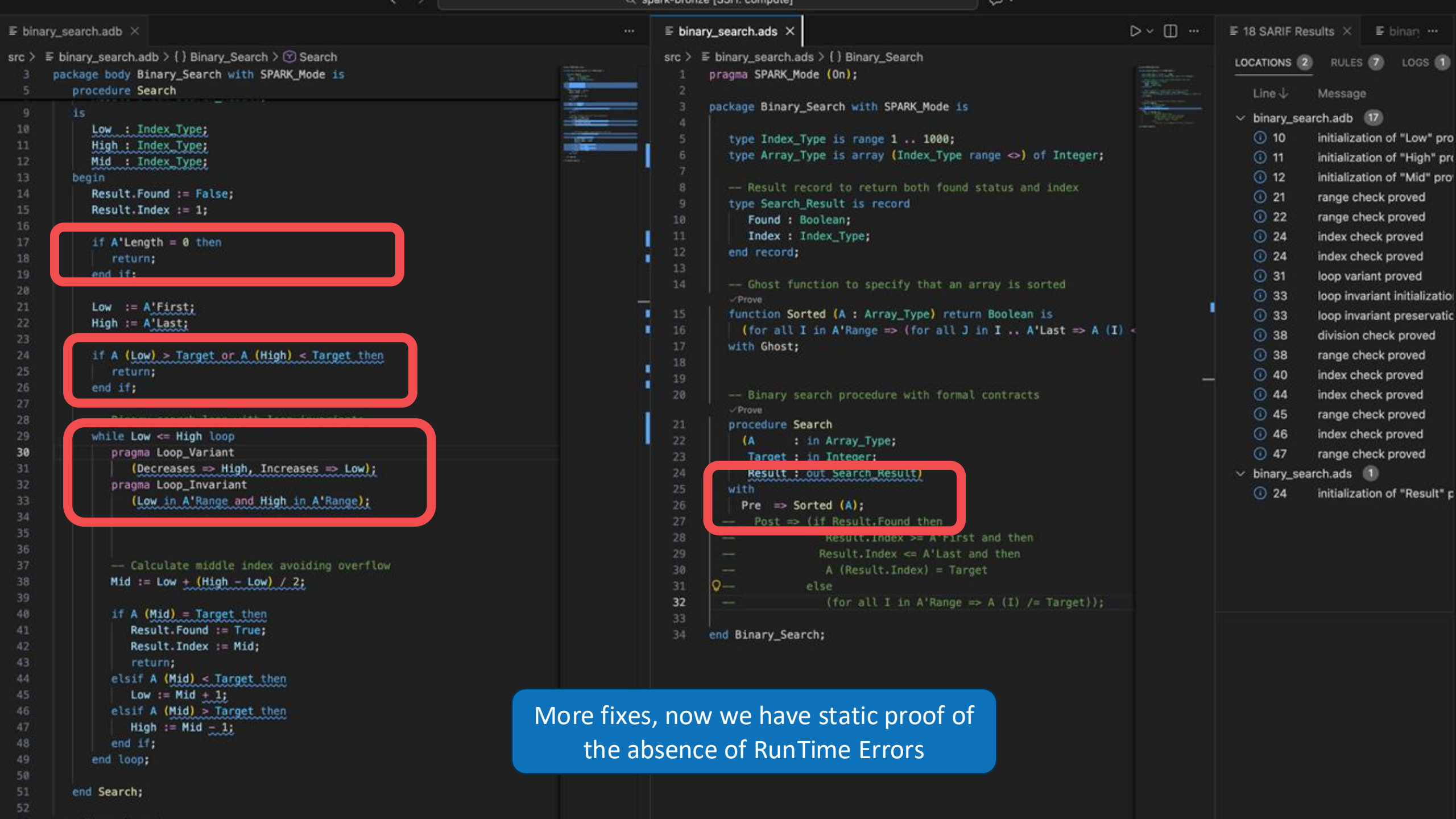
LOCATIONS 2 | RULES 4 | LOGS 1 | Filter r

Line ↓	Message
binary_search.adb 12	
10	initialization of "Low" proved
11	initialization of "High" proved
12	initialization of "Mid" proved
16	range check might fail, cannot prov
17	range check might fail, cannot prov
25	division check proved
25	range check proved
27	array index check might fail
31	index check proved
32	range check might fail, cannot prov
33	index check proved
34	range check might fail, cannot prov
binary_search.ads 1	
24	initialization of "Result" proved

INFO | ANALYSIS STEPS 0 | STACKS 0

array index check might fail

Rule Id	VC_INDEX_CHECK
Rule Name	—
Rule Description	Check that the given index i within the bounds of the arr
Level	warning
Kind	open
Baseline State	new


binary_search.adb
src > binary_search.adb > { } Binary_Search > Search
package body Binary_Search with SPARK_Mode is
procedure Search
is
Low : Index_Type;
High : Index_Type;
Mid : Index_Type;
begin
Result.Found := False;
Result.Index := 1;
if A'Length = 0 then
return;
end if;
Low := A'First;
High := A'Last;
if A (Low) > Target or A (High) < Target then
return;
end if;
while Low <= High loop
pragma Loop_Variant
(Decreases => High, Increases => Low);
pragma Loop_Invariant
(Low in A'Range and High in A'Range);
-- Calculate middle index avoiding overflow
Mid := Low + (High - Low) / 2;
if A (Mid) = Target then
Result.Found := True;
Result.Index := Mid;
return;
elsif A (Mid) < Target then
Low := Mid + 1;
elsif A (Mid) > Target then
High := Mid - 1;
end if;
end loop;
end Search;
binary_search.ads
src > binary_search.ads > { } Binary_Search
pragma SPARK_Mode (On);
package Binary_Search with SPARK_Mode is
type Index_Type is range 1 .. 1000;
type Array_Type is array (Index_Type range <>) of Integer;
-- Result record to return both found status and index
type Search_Result is record
Found : Boolean;
Index : Index_Type;
end record;
-- Ghost function to specify that an array is sorted
Prove
function Sorted (A : Array_Type) return Boolean is
(for all I in A'Range => (for all J in I .. A'Last => A (I)
with Ghost;
-- Binary search procedure with formal contracts
Prove
procedure Search
(A : in Array_Type;
Target : in Integer;
Result : out Search_Result)
with
Pre => Sorted (A);
Post => (if Result.Found then
Result.Index >= A'First and then
Result.Index <= A'Last and then
A (Result.Index) = Target
else
(for all I in A'Range => A (I) /= Target));
end Binary_Search;
18 SARIF Results
LOCATIONS 2
RULES 7
LOGS 1
Line
Message
binary_search.adb 17
10 initialization of "Low" pro
11 initialization of "High" pr
12 initialization of "Mid" pro
21 range check proved
22 range check proved
24 index check proved
24 index check proved
31 loop variant proved
33 loop invariant initializatio
33 loop invariant preservatic
38 division check proved
38 range check proved
40 index check proved
44 index check proved
45 range check proved
46 index check proved
47 range check proved
binary_search.ads 1
24 initialization of "Result"
More fixes, now we have static proof of the absence of RunTime Errors

binary_search.adb

```
src > binary_search.adb > { } Binary_Search > Search
3   package body Binary_Search with SPARK_Mode is
5      procedure Search
9      is
10        Low  : Index_Type;
11        High : Index_Type;
12        Mid  : Index_Type;
13     begin
14        Result.Found := False;
15        Result.Index := 1;
16
17        if A'Length = 0 then
18           return;
19        end if;
20
21        Low  := A'First;
22        High := A'Last;
23
24        if A (Low) > Target or A (High) < Target then
25           return;
26        end if;
27
28        -- Binary search loop with loop invariants
29        while Low <= High loop
30           pragma Loop_Variant
31              (Decreases => High, Increases => Low);
32           pragma Loop_Invariant
33
34           pragma Loop_Invariant
35              (for all I in A'Range => (if I < Low then A (I) < Target));
36           pragma Loop_Invariant
37              (for all I in A'Range => (if I > High then A (I) > Target));
38
39
40
41           -- Calculate middle index avoiding overflow
42           Mid := Low + (High - Low) / 2;
43
44           if A (Mid) = Target then
45              Result.Found := True;
46              Result.Index := Mid;
47              return;
48           elsif A (Mid) < Target then
49              Low := Mid + 1;
50           elsif A (Mid) > Target then
51              High := Mid - 1;
52           end if;
```

binary_search.ads

```
src > binary_search.ads > { } Binary_Search
1   pragma SPARK_Mode (On);
2
3   package Binary_Search with SPARK_Mode is
4
5      type Index_Type is range 1 .. 1000;
6      type Array_Type is array (Index_Type range <>) of Integer;
7
8      -- Result record to return both found status and index
9      type Search_Result is record
10        Found : Boolean;
11        Index : Index_Type;
12     end record;
13
14     -- Ghost function to specify that an array is sorted
       ✓Prove
15     function Sorted (A : Array_Type) return Boolean is
16       (for all I in A'Range => (for all J in I .. A'Last => A (I) <
17     with Ghost;
18
19
20     -- Binary search procedure with formal contracts
       ✓Prove
21     procedure Search
22       (A      : in Array_Type;
23        Target : in Integer;
24        Result : out Search_Result)
25     with
26
27        Post => (if Result.Found then
28                    Result.Index >= A'First and then
29                   Result.Index <= A'Last and then
30                    A (Result.Index) = Target
31                 else
32                    (for all I in A'Range => A (I) /= Target));
33
34  end Binary_Search;
```

27 SARIF Results

LOCATIONS 2 RULES 8 LOGS 1

Line	Message
binary_search.adb 23	
10	initialization of "Low" pro
11	initialization of "High" pr
12	initialization of "Mid" pro
21	range check proved
22	range check proved
24	index check proved
24	index check proved
31	loop variant proved
33	loop invariant initializatio
33	loop invariant preservatic
35	index check proved
35	loop invariant initializatio
35	loop invariant preservatic
37	index check proved
37	loop invariant initializatio
37	loop invariant preservatic
42	division check proved
42	range check proved
44	index check proved
48	index check proved
49	range check proved
50	index check proved
51	range check proved
binary_search.ads 4	

Finally, we can enable the Post condition and help with another invariant.
=> Proof of Functional correctness

One More Thing: Representation Clauses

Safe Hardware Access

From the C based SPI handling on a Raspberry Pi in Linux

```
#define SPI1_BASE   0x40013000U
#define SPI_CR1     (*(volatile uint16_t *)(SPI1_BASE))

#define SPI_CR
#define SPI_CR
#define SPI_CR
#define SPI_CR
#define SPI_CR
#define SPI_CR
#define SPI_CR1_LSBFIRST  (1U << 7)
/* ...and so on for every bit... */

/* Configure: master mode, baud /16, enable peripheral */
SPI_CR1 = (SPI_CR1 & ~SPI_CR1_BR_Msk) | (3U << SPI_CR1_BR_Pos);
SPI_CR1 |= SPI_CR1_MSTR;
SPI_CR1 |= SPI_CR1_SPE;
```

The Ada version is more verbose, but all directly derived from the datasheet and the compiler catches mistakes

From the Ada SPARK equivalent:

```
type CPHA_Type    is (First_Edge, Second_Edge) with Size => 1;
type CPOL_Type    is (Idle_Low, Idle_High)     with Size => 1;
type Master_Type  is (Slave, Master)           with Size => 1;
type Baud_Type    is mod 2**3                  with Size => 3;

type SPI_CR1_Register is record
   CPHA      : CPHA_Type;
   CPOL      : CPOL_Type;
   MSTR      : Master_Type;
   BR        : Baud_Type;
   SPE       : Boolean;
   LSBFIRST  : Boolean;
   SSI       : Boolean;
   SSM       : Boolean;
   RXONLY    : Boolean;
   CRCL      : Boolean;
   CRCNEXT   : Boolean;
   CRCEN     : Boolean;
   BIDIOE    : Boolean;
   BIDIMODE  : Boolean;
end record
   with Size => 16, Bit_Order => System.Low_Order_First;


CR1 : SPI_CR1_Register
   with Address => System'To_Address (16#4001_3000#), Volatile;

-- Use site: type-safe, self-documenting
CR1.MSTR := Master;
CR1.BR   := 2#011#;   -- fPCLK/16
CR1.SPE  := True;
```

```
for SPI_CR1_Register use record
   CPHA      at 0 range  0 ..  0;
   CPOL      at 0 range  1 ..  1;
   MSTR      at 0 range  2 ..  2;
   BR        at 0 range  3 ..  5;
   SPE       at 0 range  6 ..  6;
   LSBFIRST at 0 range  7 ..  7;
   SSI       at 0 range  8 ..  8;
   SSM       at 0 range  9 ..  9;
   RXONLY    at 0 range 10 .. 10;
   CRCL      at 0 range 11 .. 11;
   CRCNEXT   at 0 range 12 .. 12;
   CRCEN     at 0 range 13 .. 13;
   BIDIOE    at 0 range 14 .. 14;
   BIDIMODE at 0 range 15 .. 15;
end record;
```

But This is Additional Work?!?

Numbers from NVIDIA Offensive Security Team

Project	Language	Timeframe	Total bugs	% of Mem Safety
OS-like software	C++	3 weeks	45	~50%
	SPARK	6 weeks	10	0%
Boot software	C++	4 weeks	17	~65%
	SPARK	4 weeks	5	0%

DEFCON 30
https://www.youtube.com/watch?v=TcIaZ9LW1WE

Try It Yourself!

Open Source

Download and learning resources

- Alire.ada.dev
- Ada-lang.io
- Learn.adacore.com

Or use the Alire skill with your LLM: https://github.com/AdaCore/skills

- `'Use the /alire skill to create a binary search application including test cases in Ada on Zephyr on the NXP FRDM-MCXA156'`
  - 11 mins, including West install and Zephyr download.
- Downloads Alire, install tools, generates the code and downloads to target

Agentic AI

The field of software engineering is changing, this includes embedded.

Agentic AI Needs Strong Signals

Help your AI help itself

Ada and SPARK language features guide the AI on the path to correctness.

It can iterate rapidly without having to rely on unit testing.

The Ada compiler and GNATprove provide extensive details in their warnings.

Two Agentic AI Workflows To Try

Generate Code, then Lift to Silver

- LLMs are very good at generating Ada
- They are also good at translating C to Ada
  - Can be done in pieces, Ada and C mix well
- Then use the /gnatprove skill to lift it to SPARK Silver

Samples:

- Linux on the Raspberry Pi with an Ada GPIO and SPI driver:
  - https://gitlab.com/mhermeling/ada-linux/-/tree/ada-spi-repclause?ref_type=heads
- Curl with several modules replaced by Ada
  - https://gitlab.com/mhermeling/ada-curl

Take Aways

#1: Please invest in memory safety for your next application

#2: Formal methods are a game changer, especially with LLMs

#3: LLMs are here to help, run a PoC and see how they can help you

#4: SPARK with it's strong typing is a key enabler

THANK YOU

www.embeddedonlineconference.com

THE SPEAKER

Mark Hermeling

https://www.linkedin.com/in/markhermeling/

https://substack.com/@markhermeling

https://www.adacore.com/blog

Head of Technical Marketing

Software Development Tools and Processes

Mark has over 25 years of experience in software development tooling, operating systems, virtualization and networking technology in high-integrity, safe and secure, embedded and real-time systems. He has worked on projects building automotive, networking, aerospace and defense and industrial devices in North America, Europe and Asia. As Head of Technical Marketing at AdaCore he drives the narrative that crosses the chasm between technical capabilities and business value. Mark is a frequent author and speaker on topics ranging from the software development lifecycle, DevSecOps to formal methods and software verification. Prior to joining AdaCore, Mark worked for CodeSecure (formerly GrammaTech), Wind River Systems, Zeligsoft and IBM Rational. He holds a Master of Science degree in Computing Science from Eindhoven University of Technology.