AdaCore WHITE PAPER

Security-Hardening Software Libraries with Ada and SPARK

AdaCore

Security-Hardening Software Libraries with Ada and SPARK

White Paper

Kyriakos Georgiou, Paul Butcher, Yannick Moy

June 4, 2021

Abstract

When it comes to providing software assurance, SPARK, a formally provable subset of Ada, is the best weapon in the arsenal to ensure the absence of data-flow and runtime errors. Furthermore, SPARK offers mechanisms to prove key integrity properties and even complete functional proof of requirements. Regardless of the apparent benefits of using SPARK in safety-critical applications, there is still some reservation in adopting the technology over traditional languages like C and C++, which offer minimum software assurance. Instead, developers in many cases rely on heavy testing approaches with the hope that all the possible bugs will be captured before application deployment. Arguments in favour of such problematic approaches include a perceived steep-learning curve for adopting SPARK, and thus, an increase in production cost, and the assumption that SPARK will significantly negatively impact the execution time of an application. These are all perceptions mainly formed due to a lack of understanding the technology. This paper takes a quantitative approach, where SPARK is put under test, and relevant metrics, such as performance and development time, are recorded to demystify the impact of adopting SPARK and expose any areas that the technology can be further improved. To achieve this a relevant to the industry embedded benchmark suite written in C, the EMBENCH, is converted to Ada and SPARK, to prove the absence of runtime errors, which are well-known sources of security vulnerabilities. Furthermore, we share our valuable experiences on best practices for hardening software libraries using SPARK.

Contents

1 Introduction

For many years, safety engineering has been the main focus of certification around critical industries, such as aviation and rail systems. This is particularly true for software. Software safety standards and guidelines, for example, the DO-178 (B,C versions) for avionics, or the CENELEC EN 50128 for rail systems, have been around for decades and went through a series of revisions. These documents are well understood and admitted by the relevant industries. In contrast, software security aspects did not accept the same amount of attention within these industries. Partially, this is because some of the possible software security vulnerabilities can be addressed through the well established safety practices.

With the take-off of cyber-physical systems and the last decade's Internet of Things (IoT) evolution [5], it was made evident that security for such systems is a major challenge [11, 7]. This is reasonable as now attackers have a much bigger playground to explore and discover vulnerabilities that can be maliciously exploited. Nowadays, it is not uncommon to come across daily news referring to newly discovered software vulnerabilities that can compromise the security of billions of embedded systems, including critical ones, such as medical applications [10].

The tremendous increase in sophisticated cyber-security related attacks, leaves no room for addressing security vulnerabilities as a byproduct or a side-effect of certifying for safety. A clear understanding of the boundaries and interactions between safety and security is needed. Aerospace is one sector which has seen a recent immediate need for security assurance, particularly where security threats have been identified as leading to safety hazards. To address this, new security standards were introduced, such as the ED-202A, titled "Airworthiness Security Process Specification" and the ED-203A, titled “Airworthiness Security Methods and Considerations”. This group of standards forms a set of objectives allocated to Security Assurance Levels (SAL) which, when met, form an Acceptable Means of Compliance (AMC) by EASA for aviation cyber-security airworthiness certification. The same is also true of the DO-326A and DO-356A standards for the Federal Aviation Authority (FAA) although, at the time of writing, the FAA has not issued rules nor an Advisory Circular listing the standards as Acceptable Means of Compliance.

The role of software-language technologies in the security and safeness of cyber-physical systems is paramount [3]. The language chosen to develop a cyber-physical system is the starting point, and probably the most critical, of building a security and assurance argument against relevant certification. Thus, the Ada programming language has enjoyed wide acceptance in the space of critical software during the last four decades. This is due to the

extensive language built-in safety features, [12], that eliminate a significant number of software vulnerabilities, commonly found in other popular languages such as *C* and *C++* [2]. SPARK technology, a formally provable subset of Ada and a set of software verification tools, takes what Ada offers to the next level. It allows to mathematically prove the correctness of information flow, freedom from runtime errors, functional correctness, and the adherence to security and safety policies [4]. Thus, SPARK is perhaps the only language technology that exhibits all the essential characteristics for developing high integrity applications that can fulfil both the safety and security requirements when certifying against the highest levels of the aerospace software standards.

In this work, the SPARK technology is put under test to demonstrate that it can be relatively easily adopted and that the massive benefits of its adoption do not come with a significant negative impact on the performance of a program. To showcase this a set of benchmarks from a relevant to the industry *C* embedded benchmark suite, namely the EMBENCH suite [34], are converted to SPARK with the aim of guaranteeing the Absence of Runtime Errors (AoRTE). Runtime errors are a well-known source of security-related vulnerabilities, with dozens of system security breaches related to them [1]. Thus, such runtime errors should be eliminated in the context of highly critical systems to avoid potential exploitations that can lead to safety issues.

The work not only achieves the above goals but in the process, a plethora of useful guidelines and best practices are offered to enable others in rapidly adopting the technology for hardening software libraries.

The rest of the paper is organized as follows. Section 2 gives a quick overview of the Ada and SPARK technologies, with a focus on their software assurance by design philosophy. Section 4 introduces the EMBENCH benchmark suite, and it analyses all the steps and best practices for converting the *C* benchmarks to SPARK and proving AoRTE. Our experimental evaluation is presented in Section 5. Finally, Section 6 concludes the paper and outlines opportunities for future work.

2 Ada and SPARK

Undoubtedly programming languages is one of the most active areas of computer science. The number of new programming languages emerging is astonishing. For example, in the last two decades, *C#*, *Scala*, *Go*, *Rust*, *TypeScript*, *Kotlin* are some of the most notable languages introduced, all aiming to fulfil the needs of rapid technological advancements in Information Communication Technologies (ICT). Having a plethora of programming languages can be seen as troublesome for the software community. Software

developers are often required to master a new programming language to fulfil the needs of a new project. Becoming proficient in a new programming language often comes with a steep learning curve, and the investment in doing so in the context of a company needs to be worthwhile.

Nevertheless, the bottom line is that the best tool should be used for the task at hand. For example, *Python*, a high-level dynamically typed and interpreted programming language, found wide acceptance due to its ease of learning and its philosophy of enabling the rapid development of application prototypes. When it comes to strict performance requirements, *C* and *C++* will be chosen over *Python* since compiled code can typically outperform interpreted code. The success and the longevity of a programming language heavily depend on how well the language is serving its existential purpose, and if there is a continuous significant demand for it.

2.1 Ada

The Ada programming language emerged in the mid-1970s, out of the needs of the US Department of Defense (DOD) and the UK's Ministry of Defence to replace the large number of programming languages used in their embedded systems with one that will cover all their needs. The main requirement was to develop an embedded, real-time programming language suitable for safe and modular programming. After Ada won the competition specifically set out by the DOD for the creation of such language, the first official ANSI standard [20] Ada version was released in 1983, generally known as *Ada 83*. Since then, Ada has been through three major revision cycles resulting in the versions Ada 95, Ada 2005, and Ada 2012, respectively. With these new ISO [41] standardised revisions, Ada managed to evolve continuously and in many cases lead the evolution of programming languages in adapting to the new requirements that stemmed out of the rapid technological advancements of ICT during the last 40 years. Today Ada is a modern programming language suitable for large scale applications that require adaptability, maintainability, portability and performance. Some of the modern characteristics of the language include:

- Full Object Oriented Programming support.
- Concurrent programming features, including support for multicore.
- Hierarchical program composition/programming in the large support.
- Generic templates.
- Encapsulation.

A new revision of the standard, expected as Ada 2022, is currently formed. Among other developments, the new standard will support fine-grained parallelism that will significantly increase the ability of the language to exploit multicore architectures.

While Ada's evolution keeps the language to modern standards, it always ensures to not diverge from its original purpose and design philosophy: to provide a language that quarantines the maximal support when it comes to the development of large scale reliable, secure and safe software. By design, Ada is able to enforce detection, prevention and elimination of safety and security bugs at an early stage of the software development life-cycle and before they become part of the executable program. In fact, over the years, the language has been enhanced with a plethora of state of the art safety and security features that made it the best candidate when it comes to the development of high-integrity software. Some of most the important features are:

- `Very strong typing`. Data intended for one purpose will not be accessed via inappropriate operations. For example, common errors arising from treating pointers as integers are prevented. New types can be introduced easily, with the benefit of preventing data usage errors.
- `Extensive compile-time and run-time checks`. The language strong typing facilitates the detection of many common software errors by either the compile- or run-time checks. These include mismatched types range violations, parameter misuses and invalid references. Run-time checks can capture a certain number of errors related to type and memory conditions that can not be checked at compile time, such as "dangling references" (a live pointer referencing an object that went out of scope). Furthermore, run-time checks can capture errors such as [9]:
  - *Buffer overflow*; indexing an array with an out-of-bounds value.
  - *Integer overflow/wraparound*; computing an integer operation whose result exceeds the maximum integer value or is less than the minimum integer value.
  - Assigning to a scalar variable a value outside the range of that variable.
  - Dereferencing a null pointer.
  - Supplying malformed input as part of an input operation.
- `Contract-based programming`. Unlike most other languages, Ada introduced contracts, such as post- and pre-conditions, as part of the language's standard syntax. Such contracts are vital for expressing

> software and verification requirements explicitly in the source code, to allow for static analyzers or run-time checks to verify that the stated requirements are met. Contracts are not a new concept. However, Ada's innovative approach to verify contracts both statically and dynamically allows for a hybrid verification; formal verification can be applied to critical application modules, and traditional testing techniques can be utilized to gain a sufficient level of software assurance on the non-critical parts of the application.

By-design Ada and its companion static analysis tools, such as CodePeer [15], can eliminate or reduce the risks related to a large number of the MITRE Common Weakness Enumeration (CWE) documented cyber-security vulnerabilities [28]. This is a well established and widely accepted by the software community list of software and hardware weaknesses which can compromise the resilience, reliability and the integrity of software. MITRE listed software issues like using unsafe pointers or improper *null* termination of strings can not exist in an Ada program, while issues like *buffer overflow* or *integer overflow* can be dynamically captured by Ada's runtime-checks and dealt with via exception handlers. A comprehensive list of the CWE's prevented by Ada can be found at [8].

2.2 SPARK

Subsetting a programming language to facilitate the development of safe and secure high-integrity applications is a well-established methodology. Traditional languages like *C* and *C++* are too large and complex, resulting in too many error-prone features, or semantics that prevent analysability [24]. Thus, the use of such languages is inappropriate for certifying software against the highest-levels of assurance of safety standards and guidelines, such as the *DO-178C* for avionics, or the *CENELEC EN 50128* for rail systems. *MISRA-C* and *MISRA-C++* are successful examples of subsetting the *C* and *C++* languages, respectively. Their main objective is to restrict the use of any language constructs that are either error-prone or a source of ambiguity. Furthermore, they defined a large number of rules to allow vendors to provide static analysis tools that can enforce some of them automatically.

Although, *Ada* by design eliminates a lot of the error-prone features that exist in *C* and *C++* and addresses a significant number of the CWE vulnerabilities, the full usage of the language may be inappropriate for certification purposes or formal verification. Ada features, such as pointers or exceptions, are hard to be analysed to provide sound verification through formal methods. By subsetting Ada, the SPARK programming language emerged, to provide the largest possible subset of Ada which is suitable for functional specification

and static verification. Thus, SPARK already builds on a solid foundation for providing software assurance since it inherits all the safety features of Ada, such as strong typing. SPARK is not only a programming language but also a verification toolset. This document refers on the latest version of SPARK, known as SPARK 2014 [18], which is based on the latest version of Ada, Ada 2012. Currently the only available implementation of SPARK is provided by Altran and AdaCore [19].

SPARK 2014 significantly benefits from the introduction of contract-based programming in Ada 2012. Ada's syntax for writing static verification statements, called assertions, is also used by SPARK. These assertions are provided at the subprogram level to enable modular analysis and testing. Ada offers a range of assertions, such as preconditions and postconditions, and SPARK provides yet further assertions to allow expressing any semantics of a program needed for formal verification. Assertions can be either proven by formal verification tools to demonstrate that they always hold, or can be executed to show that they hold for specific execution instants. Executable semantics is a powerful tool as it allows for a hybrid formal verification and test-based approach. In a sense, parts of the program can be formally verified to prove certain properties, such as the absence of runtime errors, while for other parts of the code, assertions can be tested dynamically to gain the required level of assurance.

The following Ada features are simplified or excluded from the SPARK language (as taken from [53]):

- Uses of *access types* and *allocators* must follow an ownership policy [50], so that only one access object has read-write permission to some allocated memory at any given time, or only read-only permission for that allocated memory is granted to the possible multiple access objects.

- All expressions (including function calls) must be free of side-effects. Functions with side-effects are more complex to treat logically and may lead to non-deterministic evaluation due to conflicting side-effects in sub-expressions of an enclosing expression. Functions with side-effects should be written as procedures in SPARK.

- Aliasing of names is not permitted. Aliasing may lead to unexpected interferences, in which the value denoted locally by a given name changes as the result of an update to another locally named variable. Formal verification of programs with aliasing is less precise and requires more manual work.

- The backward *goto* statement is not permitted. Backward *gotos* can

be used to create loops, which require a specific treatment in formal verification, and thus should be precisely identified.

- The use of controlled types is not permitted. Controlled types lead to the insertion of implicit calls by the compiler. Formal verification of implicit calls makes it harder for users to interact with formal verification tools, as there is no source code on which information can be reported.
- Handling of exceptions is not permitted. Exception handling gives raise to numerous interprocedural control-flow paths. Formal verification of programs with exception handlers requires tracking properties along all those paths, which is not doable precisely without a lot of manual work. But raising exceptions is allowed (see Raising Exceptions and Other Error Signalling Mechanisms at [52]).
- Unless explicitly specified as (possibly) nonreturning, subprograms should always terminate when called on inputs satisfying the subprogram precondition. While care is taken in GNATprove to reduce possibilities of unsoundness resulting from nonreturning subprograms, it is possible that axioms generated for nonreturning subprograms not specified as such may lead to unsoundness. See Nonreturning Procedures at [51].
- Generic code is not analyzed directly. Doing so would require lengthy contracts on generic parameters, and would restrict the kind of code that can be analyzed, e.g. by forcing the variables read/written by a generic subprogram parameter. Instead, instantiations of generic code are analyzed in SPARK. See Analysis of Generics at [49].

These Ada features are omitted or restricted in SPARK either because it is too hard to be supported by formal verification analysis or because they introduce ambiguity in the execution of a program making it unsuitable for formal verification.

The tools that are part of the Altran/AdaCore implementation of SPARK are packed within the *GNATprove* formal verification tool [16] and are enabled through three different usage modes [17]. The first mode of *GNATprove*, *check* mode, checks that the part of the code to be formally verified in SPARK is valid SPARK code. The remaining two modes enable two different kinds of analysis:

- `Flow analysis`: checks the initialization of variables, looks at data dependencies between inputs and outputs of subprograms, and detects unused assignments and unmodified variables. This type of analysis is typically fast.

- `Proof`: checks for the absence of runtime errors and verifies any assertions that are expressing functional properties of the code. The execution time of this analysis can vary greatly depending on factors such as the number of provers used and the complexity of the code analyzed.

2.2.1 SPARK Levels of Software Assurance

Adopting SPARK for a new project can seem like an intimidating task, especially when having no prior experience with the technology. The difficulty of adopting SPARK depends on the scope of the analysis, such as a whole project or units of a project, and the software assurance level required. Fortunately, there are well-defined guidelines on the adoption of SPARK technology that simplify the process. These guidelines are offered in the form of five levels of SPARK, which are incremental in both the effort (and potentially related costs) required to be fulfilled and the amount of software assurance they provide. The five levels are:

1. Stone level - valid SPARK
2. Bronze level - initialization and correct data flow
3. Silver level - Absence of Run-Time Errors (AoRTE)
4. Gold level - proof of key integrity properties
5. Platinum level - full functional proof of requirements

For this work, we are aiming to reach up to the third level of SPARK. With this level of SPARK, programs are guaranteed AoRTE, a major source of security vulnerabilities. This also eliminate the need for runtime checks. To move at the highest levels of SPARK requires the precise intent and specifications of the application to be known, something not always feasible for code that is not written from scratch or well documented; as is the case with the EMBENCH benchmarks conversion.

3 The EMBENCH Benchmark Suite

EMBENCH, [34] is a recently formed free and open-source embedded benchmark suite, with the aim to provide benchmarks that reflect today's embedded Internet of Things (IoT) applications needs. The initiative for establishing the benchmark suite came from Prof. David A. Patterson [47], one of the principal

figures behind the RISC-V [48] processor. The main idea is to move away from outdated, artificial benchmarks, such as the *Dhrystone* [26] and *CoreMark* [27], which are no longer representative of modern embedded applications and introduce a benchmark suite that will continuously evolve to keep up with the new trends in embedded systems [46].

The EMBENCH benchmarks were largely sourced from the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [23]. BEEBS was created as part of a research project, namely, the Machine Guided Energy Efficient Compilation (MAGEEC) project [44], supported by the Technology Strategy Board of the UK government (currently re-branded as Innovate UK [40]). BEEBS was created out of the need for a modern benchmark suite that is specially designed for exploring both the relative energy consumption and performance of embedded systems. Therefore, BEEBS supports a wide range of embedded systems and algorithms. The current version of BEEBS, [22], includes 82 benchmarks. Two main criteria were used for the selection of the benchmarks. The first one is their portability across different embedded architectures. Thus, factors such as I/O and peripherals are excluded in favour of portability. The second criterion is their relevance to today's embedded systems. Furthermore, the benchmark suite is designed to be easily extensible to support multiple architectures and new benchmarks. These characteristics were also introduced in the EMBENCH benchmark suite but with an updated and more robust build system and better mechanisms to standardise the way of measuring and comparing performance and code size between different setups and configurations, such as different compilers and architectures.

Currently, the EMBENCH includes 19 benchmarks, [33], carefully selected to be a representative range of the application space that are typically found in today's IoT. The suite's build-system supports both native (x-86, Linux) and cross compilation of the benchmarks, currently for RISC-V [48] and Arm's Cortex-M4 [55] based development boards.

4 Benchmarks Conversion and Proof

For a fair performance comparison between C and Ada/SPARK, the underlying algorithmic logic and the main program structure of the original EMBENCH benchmarks had to be retained in their new correspondent Ada/SPARK versions. This limitation can significantly impact the effective use of the unique features and capabilities of the two languages. For example, casting from one scalar type to another scalar type of smaller range, or shifting signed integers are things that you typically do not expect to be part of an Ada or SPARK program. Furthermore, there is a limitation on which level of SPARK, see Section 2, can be achieved on each benchmark that stems from the

Language	Files	Blank	Comment	Code
C	23	2860	2165	11748
Ada/SPARK	26	1515	1750	5812
C/C++ Header	9	280	333	1833
SUM	58	4655	4248	19393

Table 1: Statistics for the benchmarks' source-code (retrieved by the *cloc* Linux command line tool [25]).

direct translation of the *C* code to equivalent Ada and then SPARK code. The original code was not developed with SPARK formal verification technology in mind. This can make it challenging to preserve the original code's logic and structure while enabling the verification tools to perform at their optimum level. Finally, in many cases, the lack of sufficient design documentation, particularly in the form of comments within the benchmarks' code, make it challenging to apply SPARK contracts that capture the semantics of the code. Thus, we are only able to achieve at best the silver level of SPARK, AoRTE, which is the default level aimed for critical software.

Table 1 shows some statistics taken from the folder that includes the source code of the benchmarks, using the Linux *cloc* (count lines of code) utility. From the 19 benchmarks included in EMBENC suite, 13 were initially chosen and converted to Ada. The choice was made based on an initial manual inspection of the code to select the ones that could be translated to Ada while preserving as much as possible of the original structure and algorithmic logic. Therefore, benchmarks that were too *C*-oriented, for example heavily using macro functions, or exercising a significant amount of library calls, were left out. Another criterion for selection was the level of understanding we could get about the actual functionality and logic of the benchmarks. Many of the benchmarks do not have sufficient documentation about their functionality, or they are artificially auto-generated to exercise a particular language construct and without any real algorithmic logic. The conversion of the 13 initially selected benchmarks resulted in 26 Ada files, as shown in table 1. This includes one specification file for each benchmark (*.ads*) that mainly specifies subprograms (functions, procedures), types, and the code bindings with the *C* code of the test harness of the benchmarks, and one (*.adb*) file which provides the full body of the specified subprograms. The total number of lines of code written in Ada/SPARK is 5812.

After converting the initially selected benchmarks to Ada another two benchmarks were excluded, the *statemate* and the *cubic* benchmarks. The first benchmark is auto-generated code which extensively tests branching. To avoid the need of using a floating-point library, "*#define float int*", was

Benchmark	Level Of SPARK	No. of Submodules at Silver level
aha-mont64	Silver	all
crc32	Silver	all
edn	Silver	all
huffbench	Silver	all
matmult-int	Silver	all
nettle-aes	Silver	all
nsichneu	Silver	all
st	Bronze	3/4
ud	Bronze	0/1
minver	Bronze	1/2
nbody	Bronze	1/2

Table 2: EMBENCH benchmarks converted to SPARK and their achieved level of SPARK.

added at the top of the original source code when the benchmark was ported into the EMBENCH suite. This made the benchmark challenging to replicate its behaviour into the strongly typed Ada. The second benchmark *cubic* could be converted to Ada, however, it is not applicable for SPARK, as the language does not yet support the "*Long Long Float*" type. The remaining 11 benchmarks used for this work are listed in Table 2. The table also shows the level of SPARK achieved for each benchmark. In the case a benchmark is not at *Silver*, the number of its sub-modules that reached *Silver* level is also given.

4.1 The process of converting the Benchmarks

The *Implementation Guidance for the Adoption of SPARK manual*, [6], offers excellent guidance on how to achieve the several levels of software assurance, described in Section 2.2.1. These guidelines form the basis for this work. The following subsections describe the stepwise process followed and the several challenges and findings for each step. The purpose is not to give a comprehensive manual as this is already done in [6], but rather to give enough context for the reader to understand the process and share the practical experiences of hardening software libraries with Ada and SPARK. Also, any significant steps/findings not found in the [6], for example, when there is a new SPARK feature, are highlighted. For this work *SPARK PRO* and *GNAT PRO* versions 21.0w are used. The code development and proving was done within the *GNAT Studio* that comes with *GNAT PRO*.

4.1.1 Supporting Ada and Spark in EMBENCH

The EMBENCH suite is by design easily extensible to support multiple embedded architectures and the addition of new benchmarks. However, this is the first attempt to extend the suite to accommodate a new programming language. As a first step the benchmark suite's build-system was extended to enable the native (X86, Linux) and cross-compilation, for the STM32F4Discovery development board, of Ada and SPARK with the GNAT compiler (version *GNAT Pro 21.0w*) [35, 36]. *C* compilation with the GNAT compiler is also supported. This is critical since to have a fair performance comparison between the two versions of the benchmarks the same compiler has to be used. Command-line switches can be used to control the selection of the language and target platform for compilation. The source files of the benchmarks' code converted to Ada and SPARK are placed in the same directory with their corresponding *C* versions. Only the benchmarks' functions that are exercised by the test harness of the benchmark suite for measuring performance are translated to Ada and SPARK. These are then called through the *C*-written test harness of each benchmark using appropriate interfacing mechanisms offered by Ada [42]. This separation of the benchmark's code and test-harness is vital for the maintainability of the benchmark suite because any update in the test harness will not affect the Ada/SPARK code.

4.2 Achieving the Different SPARK Levels

Applying SPARK to gain software assurance is a modular process. This significantly reduces the effort of achieving the different levels of SPARK assurance. The modularity is mainly two-fold:

1. Being able to apply the three modes of *GNATprove* at the line, region, subprogram and module (file) level.
2. Each level of SPARK achieved ensures the conformance of all of its lower SPARK levels.

It is highly recommended to take a bottom-up modular approach when hardening software libraries with SPARK, such that before moving to a higher level of SPARK all its lower levels are achieved. Besides, smaller parts of the code, with the lowest amount of interaction, such as leaf-node subprograms in the call-tree of a program, should be targeted first with a gradual move towards the file level (see Section 4.2.3). This is also the approach found to work best with this work.

4.2.1 Achieving Stone Level of SPARK

This level aims to achieve valid SPARK code. For this level, the *GNATprove check* mode is used, described in Section 2.2. This typically assumes that the program at hand is already written in Ada. Our work is a bit more complicated since we had to do a translation from *C* to Ada first. The biggest tasks and challenges we faced for this stage are:

- Eliminating the use of pointer arithmetic: this was done by replacing the pointer arithmetic with array indexing. Changes to eliminate the use of pointer arithmetic were the most challenging and error-prone, thus special care is needed to avoid introducing bugs or translating into a wrong implementation; a bug-free Ada code that does not behave as the original *C* code.
- Enforcing the strong typing rules of Ada: many of the benchmarks included a large number of implicit casting. In the cases of casting between compatible types, [13], explicit casting is used in Ada. One benchmark, *edn*, is using as part of its underline algorithmic logic a cast from *long integer* to *short integer*. Because this results in truncation, it is not acceptable by SPARK. To accommodate this in SPARK an Ada *expression function*, [14], was developed to implement the equivalent casting. Within Ada an *expression function* is a function whose implementation can be given in a single expression, making it amenable to static analysis. Figure 1a, shows the implementation of our expression function in Ada 2012. Figure 1b, shows the same implementation but with the *declare* feature that will be released with Ada 2022. This feature can help to save execution time and makes the code more readable and maintainable, as demonstrated in Figure 1.
- Replacing *C*-macro functions with Ada's *expression functions*, whenever possible, or otherwise with Ada procedures.
- Rewriting functions with side effects into Ada procedures, using *out* mode parameters.
- Introducing specifications for subprograms, functions and procedures.
- Some of the benchmarks use shifting of signed numbers, which is not supported in Ada. To implement this we used the division by powers of two.
- Introducing SPARK parameter modes: one of the main differences between *C* and Ada/SPARK is the use of parameter modes. Ada is

```
1  function Long_To_Short (v : long) return short is
2   (if v mod ((long (short'Last) + 1) * 2) < (long (short'Last) + 1) then
3       short (v mod (long (short'Last) + 1))
4    else
5      short (v mod (long (short'Last) + 1) + long (short'First)));
```

(a) Expression function implementing *long int* to *short int* casting in Ada 2012.

```
1  function Long_To_Short (v : long) return short is
2   (declare
3    short_max : constant long := long (short'Last) + 1;
4    test_value : constant long := v mod (short_max * 2);
5    begin
6     (if test_value < short_max then
7        short (test_value)
8     else
9        short (test_value - 2 * short_max)));
```

(b) Expression function implementing *long int* to *short int* casting in Ada 2022. In the future release of Ada 2022, *declare* region will be allowed in expression functions which can help to save execution time and makes the code more readable and maintainable, as demonstrated in the above example. Note that the second version is not a direct translation of the first one, but rather each version is a variant to achieve the best possible performance with or with out the use of the *declare* region feature.

Figure 1: Implementations of *long int* to *short int* casting as expression function in SPARK, using Ada 2012, (a), and Ada 2022, (b).

Parameter Mode	Usage
in	Parameter can only be read, not written
out	Parameter can be written to, then read
in out	Parameter can be both read and written

Table 3: Ada's parameter modes [17].

taking a different concept to the traditional mechanisms of parameter passing by value or by reference. Instead, it utilizes keywords that specify the direction of data flow for each parameter. This is part of Ada's philosophy to avoid common issues with pointers and dynamic memory-allocation (such as dangling pointers or memory leakage). Furthermore, the explicit definition of the mode for each parameter serves the documentation of the usage of a subprogram and allows the compiler and flow analysis stage to warn in the case of misuse. The possible parameter modes are shown in Table 3.

4.2.2 Achieving Bronze Level of SPARK

This level guarantees the absence of several defects such as reads of uninitialized variables, possible interference between parameters and global variables, and unintended access to global variables. For this level, *GNATprove*'s *flow analysis* mode is used, described in Section 2.2. The analysis does not require any user annotations (contracts) and is always guaranteed to complete. The user should be aware of the, stronger than Ada's, initialization policy that SPARK is enforcing to be able to understand and resolve issues reported by flow analysis. This strongest policy facilitates a fast and scalable flow analysis by *GNATprove* that ensures appropriate initialization of data. The policy is documented in [29], and a quick intro of it is also available in the "*4.2.1 Strong Data Initialization Policy*" section of [6].

The most important tasks performed or issues found at this stage are:

- Initialization and aliasing issues were dealt with by the approaches described in [6]. Further, discussion about Initialization related issues will take place in Section 4.2.3, where the new *Relaxed_Initialization* feature of SPARK is demonstrated.

- Although no contracts are needed for this analysis, *Global* contracts are used to specify which global variables are read and/or written by subprograms. This is a good practice as they serve as a documentation of the intent of the programmer, and thus, increase the maintainability of code. Furthermore, *GNATprove* checks this automatically and spots any discrepancies between the implementation and the specifications [6]. Further documentation and examples regarding *Global* contracts and other types of useful contracts, such as the *Depends* contract, can be found here [43]. One important note is that once you specified a *Global* contract, it must be complete in the sense that all global variables accessed by the subprogram have to be included with their valid mode. Flow analysis will also warn in case of incomplete contracts. Examples of *Global* contract usage can be seen in Figure 2a, lines 9 and 16.

- The flow analysis will provide feedback over the correct use of each subprogram parameter's stated mode. This checks that each specified parameter mode corresponds to the actual usage of that parameter in the subprogram's body; for example flow analysis will report errors such as an uninitialized variable is using the *in* mode. Such errors should be easy to fix following Table 5, which summarizes SPARK's valid parameter modes as a function of whether reads and writes are done to the parameter [17].

Benchmark	Issues Captured by Flow Analysis
minver	1. “U” and “V” local variables of the “minver” procedure are part of four unused assignments. 2. Removing the unused assignments of “U”, “V” makes the same variables and the “col” parameter unused.
aha-mont64	The “uhi” local variable of the “montmul” function has an unused assignment.

Table 4: Issues discovered for benchmarks by flow analysis.

Parameter mode	Initial value read	Written on some path	Written on every path
in	X		
in out	X	X	
in out	X		X
in out		X	
out			X

Table 5: SPARK’s valid parameter modes as a function of whether reads and writes are done to the parameter [17].

- Flow analysis can also generate other valuable warnings such as dead code, initializations that have no effect, unused assignments and statements that have no effect. Such issues are indicators of possibly incomplete or wrong implementation and should be investigated further before addressing them. Table 4 lists all the issues captured by flow analysis. After further investigation, we found that the *minver* had possibly an incomplete or wrong implementation. The benchmark is implementing a matrix inverse algorithm. When manually calculating the determinant and the inverted matrix, for the matrix tested in the benchmark, it was found that our results did not match with the benchmark’s actual output. This prompted us to investigate why the benchmark’s testing mechanism, built-in to the benchmark suite, was accepting the result of the original *C* code as correct. Looking at the test’s expected result that the actual result was checked against, we found that it was also wrong and equal to the actual result. This demonstrates that testing is inherently limited as a means of guaranteeing that a program is conforming to its functional specifications. Thus, SPARK levels 4 and 5 should be used when aiming for ensuring functional correctness. The issues found will be reported to the EMBENCH community.

Bronze level of SPARK was achieved for all the benchmarks.

```
1  MAX : constant Integer := 100;
2  subtype Index is Integer range 0 .. MAX-1;
3  subtype Custom_Float is Long_Float range 0.0 .. 100.0;
4  Type Custom_Float_Array is array (Index) of Custom_Float;
5  Seed : Natural range 0 .. 8094;
6
7  procedure Init_Seed
8    with
9      Global => (Output => (Seed)),
10     Export => True,
11     Convention => C,
12     External_Name => "InitSeed";
13
14 procedure Initialize (Data : out Custom_Float_Array)
15   with
16     Global => (In_Out => (Seed)),
17     Relaxed_Initialization => Data,
18     Post => Data'Initialized and then
19     (for all J in Index'Range => Data (J) in 0.0 .. 100.0),
20     Export => True,
21     Convention => C,
22     External_Name => "Initialize";
```

(a) Part of the *st* benchmark's specification file. Lines 1-5 demonstrate the use of precise types to assist *GNATprove* to prove the AoRTE.

```
1  -- Initializes the seed used in the random number generator.
2  procedure Init_Seed is
3  begin
4    Seed := 0;
5  end Init_Seed;
6  -- Intializes the given array with random integers.
7  procedure Initialize (Data : out Custom_Float_Array) is
8  begin
9    for i in Index loop
10     Seed := ((Seed * 133) + 81) mod 8095;
11     Data (i) := Custom_Float (Long_Float (i) + Long_Float (Seed) / 8095.0);
12     pragma Loop_Invariant
13       (for all J in Index'First .. i =>
14         Data (J) in 0.0 .. 100.0 and Data (j)'Initialized);
15   end loop;
16 end Initialize;
```

(b) Part of the *st* benchmark's implementation file. The definitions of precise types to assist *GNATprove* to prove the AoRTE are derived out of the semantics of the calculations that affect each variable, for example, *Seed*, *Index* and *Data*.

Figure 2: Part of the *st* benchmark's code to demonstrate the use of precise types.

4.2.3 Achieving Silver Level of SPARK

This stage aims to guarantee the absence of runtime errors, and it is the minimum desirable level for critical software. This level is the most challenging to achieve compared to the previous ones because a good understanding of the semantics of the benchmarks' code is required.

To guarantee AoRTE, it is crucial to understand what *GNATprove* aims to achieve when applying the prove mode and what is the role of the user in guiding this process:

- *GNATprove*'s aim is to compute all possible values of variables to guarantee the AoRTE.
- User's role is two-fold:
    1. To provide extra information into the program, needed by *GNATprove*'s formal verification to successfully complete a proof. Such information must be embodied in precise-enough types and contracts related to the semantics of the code by the user.
    2. To interact with the *GNATprove* tool to assist with achieving its aim. This interaction should be done again in a bottom-up, modular fashion to make the proving process as simple as possible. Firstly, the user should start the proving process by selecting subprograms that are leaf-nodes in the call-tree of the program and then moving upwards, each time focusing on the unproven programs with the fewest calls. Secondly, the user should start by applying the least costly level of analysis offered by *GNATprove*. Then, when more information is provided into the program to support the proving and the current level of the analysis does not seem adequate for automatic proving, the user should gradually scale up the analysis power used. This is a human-driven process where experience is a significant factor in judging when a level of analysis is sufficient or not. Experimenting is the best way to rapidly accumulate such expertise. To support this process *GNATprove* offers four default levels of proving, [38], where each higher proof-level is increasing the precision of the analysis, mainly either by allowing more provers to be used or more analysis time for the provers.

As expected, when running *GNATprove*'s prove mode for the first time on the benchmarks that are already at the *Bronze* level of SPARK, it resulted in a large number of possible runtime errors. These were mainly related to array index out of bounds, arithmetic overflow, value out of range, and division by zero. At this stage, the user has to play his role, as described above. This

is a process that significantly gets easier as the user accumulates more experience in assisting the *GNATprove* with the proving of AoRTE.

The following steps proved to be sufficient to eliminate the majority of unproven checks raised by the tool, for the majority of the benchmarks, applied in the following given order and always following the bottom-up, modular approach described earlier:

- The use of sufficiently precise types: this is essential information for *GNATprove*, as it depends on the ranges of *Scalar* types to prove the absence of several runtime errors such as overflows. At *Bronze* level, all the benchmarks were still using standard scalar types such as Integer and Float, giving no information about the range of the data manipulated to the provers. This inevitably resulted in *GNATprove* giving an overflow check message for the majority of the arithmetic operations. Other related emitted errors can be array-index or divide-by-zero checks. The majority of the standard types used had to be replaced with more specific types or subtypes, with suitable ranges, as a first step of allowing the provers to address these unproven checks. An example of this is given in Figure 2a, lines number 1-5, where precise types are given for the *Seed*, *Index*, and *Data* variables. These precise types are derived through an understanding of the semantics of the code that affect the values of these variables; see *Init_Seed* and *Initialize* subprograms given in Figure 2b. Providing this kind of information, was sufficient to eliminate the false alarms regarding overflows and array index out of bounds for the *Initialize* procedure.

- The use of subprogram-level contracts that can further assist *GNATprove* to prove the AoRTE: three kinds of contracts were sufficient to eliminate the majority of false alarms, namely the precondition, postcondition, and loop-invariant contracts. It is important to understand that these contracts are mechanisms that provide information which can be propagated by *GNATprove* along the flow of the program, both backwards and forward, to allow the tool to prove certain properties in the context of the program as a whole. For example, preconditions and postconditions are acting as the communication points with the outside world; they can communicate relevant properties to their callers. Similarly, loop invariants can be used to prove AoRTE inside and after a loop. More specifically:

  - `Preconditions` specify in which context the program may be called. They can also be used for defensive code. An example of a precondition is given in Figure 3.

```
1        function modul64
2          (x : Unsigned_64;
3           y : Unsigned_64;
4           z : Unsigned_64)
5           return Unsigned_64
6          with
7            Global => null,
8            Pre => x < z,
9            Export => True,
10           Convention => C,
11           External_Name => "modul64";
```

Figure 3: The SPARK specification of *modul64* function of the *aha-mont64* benchmark demonstrates the use of a precondition to specify a property that has to hold upon each function call. The property is specified in the comments of the original code [31].

- `Postconditions` specify what can be guaranteed about the result of a subprogram. An example of a postcondition is given in Figure 2a at line 18. The postcondition's proof conveys the information that after each call of *Initialize*, *Data* array is guaranteed to be initialized.
- `Loop invariants` provide valuable information about the states of variables that are updated within a loop to allow *GNATprove* to prove the AoRTE within the loop. Examples of loop-invariant usage can be seen in Listing 1 at lines 56 and 58, for the *jpegdct* procedure of the *edn* benchmark. The loop invariant is true at each iteration of the loop. More information about the use of loop invariants can be found in the SPARK manual [39]. A short and very comprehensive explanation on the value and usage of loop invariants can be found here [39].

With the use of more precise types and the described three types of contracts, the majority of unproven checks were eliminated for most of the benchmarks. The rest of the section will discuss some less trivial cases and how they were addressed.

`Initialization check might fail`:
As discussed in Section 4.2.2, SPARK has a stronger initialization policy than Ada. The SPARK manual summarizes the consequences of the policy, [29], in the following sentence:

> "Hence, all inputs should be completely initialized at subprogram entry, and all outputs should be completely initialized at subprogram output. Similarly, all objects should be completely initialized

> when read (e.g. inside subprograms), at the exception of record subcomponents (but not array subcomponents) provided the subcomponents that are read are initialized."

This policy may be too strict in some cases. For example, it does not permit partial initialization of composite objects through different subprograms, resulting in the *GNATprove* issuing a related check. A new SPARK feature, introduced in *GNATprove* (within the SPARK Community 2020 and SPARK Pro 2021 releases) called *Relaxed_Initialization*, allows opting-out of the strict initialization policy when the user considers appropriate for specific objects and provides the mechanisms to enable the tool to prove that every access to those objects is to initialized data. A complete description of this feature can be found here [21].

Figure 4 demonstrates a possible use-case of the new *Relaxed_Initialization* feature of SPARK, from the *matmult_int* benchmark. In this example the *Res* matrix is being progressively initialized in the second nested loop, while the initialized elements of the matrix are read in the innermost nested loop, see Figure 4b lines 14 and 23, respectively. *GNATprove* will emit an *Initialization check might fail* message due to the SPARK initialization policy. To relax the initialization policy for the *Res* matrix we denote it with the *Relaxed_Initialization* aspect, see Figure 4a line 5. The *Initialized* attribute is then used within a loop invariant for each loop to express which elements of the *Res* matrix are initialized at that point of the loop, and enable *GNATprove* to verify that every access to the *Res* is on initialized data, see Figure 4b lines 5, 10 and 16.

The user should only use the above approach in cases where maximum confidence over the initialization of a specific object is needed, as it requires more effort to verify data initialization from both the user and the tool. An alternative and easier to implement solution of resolving the initialization check of the Figure 4 example is to initialize the entire *Res* matrix at the begin of the subprogram using "*Res* := (*others* => 0);". However, in this case, the latter solution had a significant performance penalty. Besides, when applying this approach, extra caution is needed to ensure that the alternated initialization is not just for the sake of verification, but also it preserves the original semantics of the code. Alternatively, the user can use the "*pragma Annotate*" to accept the check message. There are several ways to eliminate initialization checks, which exhibit different pros and cons. These are described with examples in the "*4.2 Initialization Checks*" section of [6].

`Further tips for proving`:
The ability of using SPARK to achieve the several levels of assurance comes

```
procedure Multiply (A : Input_Matrix; B : Input_Matrix; Res : out Matrix)
 with
   Global => null,
   Export => True,
   Relaxed_Initialization => Res,
   Convention => C,
   External_Name => "Multiply";
```

(a) The specification of the *Multiply* function, demonstrating the use of *Relaxed_Initialization* feature on the *Res* matrix.

```
procedure Multiply (A : Input_Matrix; B : Input_Matrix;
                    Res : out Matrix) is
begin
  for Outer in Index loop
    pragma Loop_Invariant
     (for all J in 0 .. Outer - 1 =>
       (for all K in Index =>
          Res (J, K)'Initialized));
    for Inner in Index loop
      pragma Loop_Invariant
       (for all J in 0 .. Outer =>
         (for all K in 0 .. Inner - 1 =>
            Res (J, K)'Initialized));
      Res (Outer, Inner) := 0;
      for I in Index loop
        pragma Loop_Invariant
         (for all J in 0 .. Outer =>
           (for all K in 0 .. Inner =>
             Res (J, K)'Initialized) and then
             Res (Outer, Inner) <=
             Random_Range_Long_Int'Last**2 * Long_Integer (I));
        Res (Outer, Inner) :=
          Res (Outer, Inner) + A (Outer, I) * B (I, Inner);
      end loop;
    end loop;
  end loop;
end Multiply;
```

(b) The implementation of the *Multiply* procedure, demonstrating the use of the *Initialized* attribute in loop invariants to enable *GNATprove* to prove that every access to *Res* matrix is to initialized data.

Figure 4: An example of the new *Relaxed_Initialization* feature of SPARK on the *Multiply* procedure of the *matmult_int* benchmark.

partially from reading the manuals and partially from accumulating practical experience. The following list is a collection of useful techniques built up over the duration of this exercise:

- *Using a loop invariant to stop GNATProve from unrolling a loop*: depending on the number of iterations (currently the threshold is 20) *GNATprove* may try to unroll a loop as it can help with the proving. In cases of loops with large bodies or many levels of nested loops unrolling might result in complex formulas that provers can not prove or extend the analysis time and memory usage significantly. This was the case for the example in Figure 4, before the introduction of loop invariants. Loop invariants prevent the unrolling of a loop by the *GNATprove*'s analysis. Thus, a good mechanism to stop unrolling when it seems reasonable is to use a *"pragma Loop_Invariant (true)"* which has no other effect except disallowing the unrolling of a loop.
- *Breaking down composite objects*: when both reads and writes are exercised on a composite type within a loop, it can be very challenging for *GNATprove* to cope with the analysis of the values flowing in each element of the composite type. For example, in the Listing 1 line four, the "`tx`", with `x` between 0-11, scalar variables are used. Instead of these scalars, the original *C* benchmark was using an array "`t`" of size 12. The use of an array, in this case, makes the proof impractical, as the prover has to unfold all updates to the array to discover the value flowing in a particular array cell. Therefore, the replacement of the array in a flat sequence of 12 scalar variables allowed the *CodePeer* static analysis tool of *GNATprove* to assist the formal verification tools (provers) with the completion of the proof. Knowing when to apply such an approach depends on the context and the level of experience of the user. A good indication of when to break down the use of composite types is when they are used for both read and write operations in the context of a complex algorithm. Writing SPARK from scratch rather than directly translating *C* code to SPARK should avoid such issues.
- *Breaking down large floating-point expressions*: in some cases breaking down large floating-point expressions to smaller equivalent ones helps the prover reason over the intermediate outcomes and enables further verification. An example of this can be seen in Figure 5b, at lines 12 and 13. The original code was using a single expression to implement the same calculations: "`diffs := diffs + (Data (i) - Mean) * (Data (i) - Mean);`". Breaking down the calculation to smaller intermediate steps and introducing precise types for the variables that store the intermediate results, such as `Long_Float_Temp`

introduced at line 6 of our example, helps *GNATprove* with the proving of AoRTE.

- *Manual proof using SPARK lemmas*: several arithmetic properties involving multiplication, division and modulo operations, such as the monotonicity of floating-point multiplication, are difficult to be proven automatically in some cases. A lot of these properties can be manually proven by simply calling the appropriate SPARK *Lemmas* from the *SPARK Lemma Library* [54]. An example of using the library can be found here [45]. Furthermore, the user can create custom lemmas to prove any properties not covered in the *SPARK Lemma Library*. An example can be seen at line 6 of Figure 5a where a custom lemma is introduced and used at line 18 of Figure 5b. This lemma expresses additional properties about the possible range of the return value of the *Sqrt* library function, based on the range of the type of the parameter used when calling the function. Thus, this extra property allows *GNATprove* to verify the AoRTE for line 19 of Figure 5b.
- *The use of executable assertions*: this is one of the most powerful features offered by Ada2012. The user can specify contracts, such as preconditions, postconditions and loop invariants, and then execute them to examine that they hold under specific instances of the program. This is very useful feedback in the process of proving AoRTE.

Automatic proving may not always be achievable due to limitations of the heuristic techniques used in automatic provers. These limitations are usually related to non-linear integer arithmetic (such as division and modulo) and floating-point arithmetic. More advanced steps can be taken to investigate unproven checks as described in the *How to Investigate Unproved Checks* section of the SPARK user's guide [37].

Applying all the techniques described in this section was sufficient to prove complete AoRTE for 7 out of the 11 benchmarks, and for five out of the nine total number of subprograms for the four remaining benchmarks (see Table 2). The remaining three subprograms from the *st*, *ud* and *nbody* benchmarks, for which *Silver* level of SPARK was not achieved, include either division operations and/or floating-point arithmetic. Thus, for these subprograms, a deeper understanding of their code is needed. This was difficult to achieve in the absence of sufficient documentation. Moreover, possible extensive code modifications to make these subprograms more amenable to static analysis and formal verification will make them less relevant for performance comparisons against their original *C* versions. In the case of the *minver* benchmark, which implements a matrix inversion algorithm, when the original benchmark's results were tested manually, they were found to be

```
1  subtype Custom_Float is Long_Float range 0.0 .. 100.0;
2  subtype Custom_Float_Var is
3    Long_Float range 0.0 .. (Custom_Float'Last**2);
4  Type Custom_Float_Array is array (Index) of Custom_Float;
5
6  procedure Lemma_Sqrt_Extra_Properties (X : Custom_Float_Var)
7    with
8     Global => null,
9     Post => (if X in Custom_Float_Var
10             then Sqrt (X) in 0.0 .. Custom_Float'Last);
```

(a) Part of the *st* benchmark specification.

```
1  procedure Calc_Var_Stddev (Data : Custom_Float_Array;
2                             Mean : Custom_Float;
3                             Var : out Custom_Float_Var;
4                             Stddev : out Custom_Float)
5  is
6    subtype Long_Float_Temp is Long_Float range 0.0 .. Custom_Float'Last**2;
7    Temp : Long_Float_Temp;
8    diffs : Long_Float range
9      0.0 .. Long_Float_Temp'Last * Long_Float (MAX) := 0.0;
10 begin
11   for i in Index loop
12     Temp := (Data (i) - Mean) * (Data (i) - Mean);
13     diffs := diffs + Temp;
14     pragma Loop_Invariant
15       (Diffs in 0.0 .. Custom_Float'Last**2 * Long_Float (I+1));
16   end loop;
17   Var := diffs / Long_Float (MAX);
18   Lemma_Sqrt_Extra_Properties(var);
19   Stddev := Sqrt (Var);
20 end Calc_Var_Stddev;
```

(b) The implementation of the *Calc_Var_Stddev* from the *st*.

Figure 5: An example of breaking up a floating-point expression to simpler ones to assist *GNATprove* with verifying AoRTE on the *Calc_Var_Stddev* procedure of the *st* benchmark. Also, the example demonstrates the use of a user-defined lemma to express the possible range of the return value of the *Sqrt* function at line 19.

wrong, meaning that the implementation contains functional errors. The flow analysis also indicated a possible incorrect or wrong implementation as it detected several unused assignments in the original code (see Section 4.2.2). Thus, we did not proceed to verify this benchmark further.

```
   -- JPEG Discrete Cosine Transform
   procedure jpegdct (d : in out Short_Array;
                      r : Short_Array) is
     t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11 : long;
     l : Integer range 0 .. 64 := 0;
     k : Integer range 1 .. 8 := 1;
     m : Integer range 0 .. 3 := 0;
     n : Shift_Range := 13;
     p : Integer range 1 .. 8 := 8;
   begin
     for j in 0 .. 1 loop
       k := k + 7 * j;
       m := m + 3 * j;
       n := n + 3 * j;
       p := p - 7 * j;
       l := l - 64 * j;
       for i in 1 .. 8 loop
         t0 := long (d (l)) + long (d (l + (k * 7)));
         t1 := long (d (l + k)) + long (d (l + (k * 6)));
         t2 := long (d (l + (k * 2))) + long (d (l + (k * 5)));
         t3 := long (d (l + (k * 3))) + long (d (l + (k * 4)));
         t4 := long (d (l + (k * 3))) - long (d (l + (k * 4)));
         t5 := long (d (l + (k * 2))) - long (d (l + (k * 5)));
         t6 := long (d (l + k)) - long (d (l + (k * 6)));
         t7 := long (d (l)) - long (d (l + (k * 7)));
         t8 := t0 + t3;
         t9 := t0 - t3;
         t10 := t1 + t2;
         t11 := t1 - t2;
         d (l) := Long_To_Short ((t8 + t10) / long (2**m));
         d (l + (4 * k)) := Long_To_Short ((t8 - t10)
          / long (2**m));
         t8 := long (Long_To_Short (t11 + t9)) * long (r (10));
         d (l + (2 * k)) := Long_To_Short (t8 +
          long (Long_To_Short ((t9 * long (r (9))) /
          long (2**n))));
         d (l + (6 * k)) := Long_To_Short (t8 +
          long (Long_To_Short ((t11 *
          long (r (11))) / long (2 * n))));
         t0 := long (Long_To_Short (t4 + t7)) * long (r (2));
         t1 := long (Long_To_Short (t5 + t6)) * long (r (0));
         t2 := t4 + t6;
         t3 := t5 + t7;
         t8 := long (Long_To_Short (t2 + t3)) * long (r (8));
         t2 := long (Long_To_Short (t2)) * long (r (1)) + t8;
         t3 := long (Long_To_Short (t3)) * long (r (3)) + t8;
         d (l + (7 * k)) := Right_Shift_Short_Constraint
          (Long_To_Short (t4 * long (r (4)) + t0 + t2), n);
         d (l + (5 * k)) := Right_Shift_Short_Constraint
          (Long_To_Short (t5 * long (r (6)) + t1 + t3), n);
         d (l + (3 * k)) := Right_Shift_Short_Constraint
          (Long_To_Short (t6 * long (r (5)) + t1 + t2), n);
         d (l + (1 * k)) := Right_Shift_Short_Constraint
          (Long_To_Short (t7 * long (r (7)) + t0 + t3), n);
         l := l + p;
         pragma Loop_Invariant (l = l'Loop_Entry + p * i);
```

```
57          end loop;
58          pragma Loop_Invariant
59            (n = n'Loop_Entry + 3 * j and m = m'Loop_Entry + 3 * j
60             and k = k'Loop_Entry + 7 * j and p = p'Loop_Entry - 7 * j
61             and l = l'Loop_Entry + p * 8);
62       end loop;
63    end jpegdct;
```

Listing 1: The SPARK implementation of *jpegdct* procedure of the *edn* benchmark.

5 Evaluation

This section deals with the performance evaluation, the time needed for the completion of the SPARK related tasks, and some issues found and improved within the SPARK technology. Furthermore, we highlight from which future enhancements SPARK technology will potentially benefit the most, based on our experience.

5.1 Performance Evaluation

The performance evaluation was done on an STM32F4DISCOVERY development board [30]. The board is equipped with a 32-bit ARM Cortex-M4 with FPU core, 1-Mbyte Flash memory, 192-Kbyte RAM, and it can run at a maximum frequency of 168MHz. The ARM Cortex-M4 and the RISK-V 32-bit processors are the two embedded processors currently supported in the EMBENCH benchmark suite due to their popularity in the embedded industry. The benchmarks, both in C and SPARK, were compiled with the *GNAT Pro 21.0w* compiler using the *-O2* optimization flag and with link-time optimizations enabled. The link-time optimizations were essential to allow a fairer comparison between the C and the SPARK versions, as the SPARK code for each benchmark is separated from its test harness source file while the C benchmarks' code lives in the same file as their test-harness. Thus, inlining of the benchmark code was feasible in the C versions and not in the SPARK versions. By enabling link-time optimizations, inlining was also enabled for the SPARK code similarly to the C code. Furthermore, Ada's runtime checks where disabled, using the *-gnaptp* flag, to make the performance comparison fair with C. Nevertheless, subprograms proven at SPARK *Silver* level come with an AoRTE guarantee, and thus runtime checks can be safely disabled. This is desirable when certifying at the highest levels of software assurance of safety standards, such as the *DO-178C* for avionics, or the *CENELEC EN 50128* for rail systems. This is equally applicable when certifying against

Benchmark	Level of SPARK	SPARK vs C (Performance)
aha-mont64	Silver	-24.07%
crc32	Silver	0.00%
edn	Silver	8.56%
huffbench	Silver	2.54%
matmult-int	Silver	4.27%
nettle-aes	Silver	10.91%
nsichneu	Silver	7.55%
st	Bronze	-16.70%
ud	Bronze	9.49%
minver	Bronze	-0.66%
nbody	Bronze	38.83%

Table 6: Performance comparison of the C and SPARK versions of each benchmark. Note that a positive percentage represents the percentage increase in execution time for a benchmark written in SPARK when comparing to the execution time of its corresponding *C* version.

security critical standards and guidelines such as *ED-202A/ED-203A* and *DO-326A/DO-356A* for aviation.

For execution time measurements, EMBENCH already offers a built-in mechanism which is used for both C and SPARK. This utilizes the on-board timers of the STM32F4DISCOVERY board which provide accurate and repeatable measurements. More about the EMBENCH test harness and usage can be found in the benchmark's suite documentation [32].

Table 6 and Figure 6 shows the performance comparison for the C and SPARK version of benchmarks. Note that even though the *minver* original benchmark is found to be functionally incorrect, (see Section 4.2.2), the SPARK level implementation matches that behaviour. Thus, performance comparison is still valid. For the majority of the benchmarks, there is no significant sacrifice in performance when moving from C to SPARK, with the *nbody* being an outlier. This is because there were no significant intrusive modifications needed to the code to support SPARK proves. Considering the significant added value in terms of software assurance gained by the use of the SPARK technology, there is no doubt that it is a worthwhile transition. The results are now passed to the compiler team of AdaCore which investigates the causes behind any significant performance differences for the two languages and will look for further optimization opportunities.

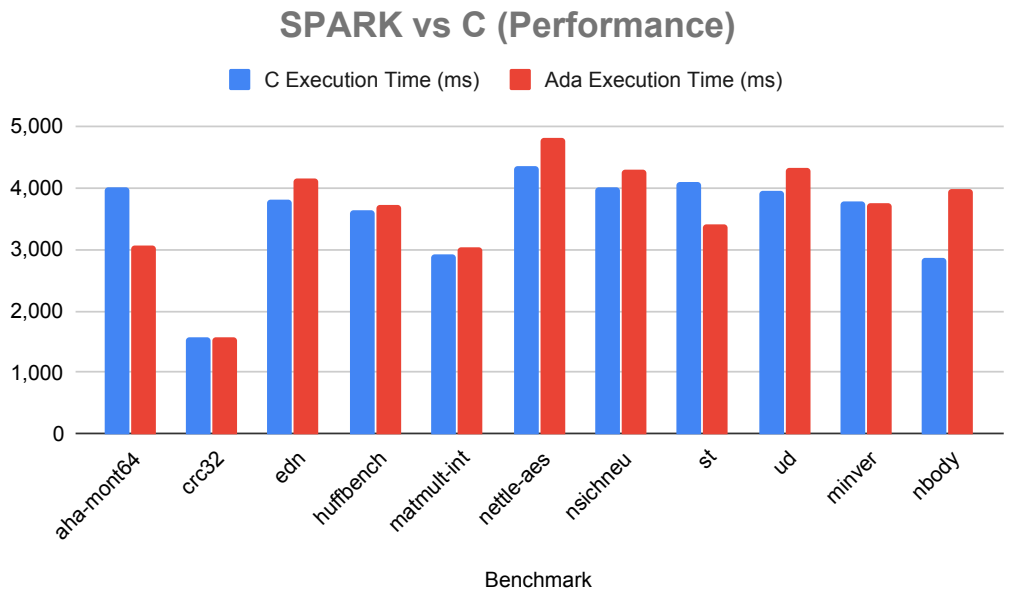

Figure 6: Performance comparison of the C and SPARK versions of each benchmark.

5.2 Effort/Time and Technology Assessment

One of the aims of this work is to evaluate the effort needed in achieving the absence of runtime errors with the SPARK technology. The completion of this work lasted around 35 working days. The level of experience with the Ada and SPARK technologies was around four months, although the engineer that carried out the work had an overall programming experience of about 15 years. Taking this into account and that silver level of SPARK was achieved within this time frame for the majority of the benchmarks, 7 out of 11, and for most of the functions for the remaining benchmarks, it is fair to say that SPARK technology is easily accessible and its adoption can yield significant benefits for hardening software libraries for security in a short time.

As discussed in Section 4, the main technical tasks involved in the completion of this work were:

1. Enabling the EMBENCH build-infrastructure to support Ada/SPARK.
2. supporting the STM32F4DISCOVERY for Ada/SPARK and C using the GNAT compiler (version GNAT Pro 21.0w).
3. Converting C to Ada.
4. Achieving the several levels of SPARK.

The biggest challenges and the most time spent was on the two last tasks. The main reasons for this are:

- Conversion from C to strongly typed Ada is hard when there is no sufficient intuition about the original code's intent.
- In many cases, unconventional ways of coding were adopted in the C code, for example, shift operations on signed numbers.
- Keeping the same implementation logic between C and Ada/SPARK for the sake of performance comparison reduces the ability to apply the SPARK technology to the highest levels of assurance (levels 3, 4 and 5) since the original code was not designed with formal verification in mind.
- A clear understanding of the code semantics is needed to achieve the silver level of SPARK. In our case, applying the silver level of SPARK, which guarantees the absence of runtime errors, was limited by the lack of proper documentation of the functional specifications for some benchmarks.

When writing a SPARK program from scratch and having clear functional specifications up-front, reaching *Silver* level should be relatively easier than what we experienced in this work. Furthermore, as engineers keep accumulating experience with the SPARK technology, the time and cost saved in the long term on certifying code against safety and security standards can be significantly lower compared to alternative approaches, such as software testing [4].

Although the Altran/Adacore implementation of SPARK 2014 significantly made the use of SPARK more accessible to developers without background knowledge in formal verification, there are still areas within the technology that could benefit from further improvement. The primary limitations identified from this work are associated with proving code that involves non-linear integer arithmetic (such as division and modulo) and floating-point arithmetic. In fact, these limitations do not stem from the SPARK technology itself but from the fact that prover technology, utilized by SPARK, has fundamental limitations in dealing with non-linear arithmetic and floating-point. Nevertheless, more can be done from the SPARK side of things to enable the provers to perform better with floating-point. Thus, work scheduled in the context of W3.5 will create SPARK floating-point mathematical libraries that will improve the floating-point proving capabilities of SPARK.

Finally, this work identified a few improvements and issues within the SPARK technology. For example, the EMBENCH benchmarks were some of

the early adopters of the "*Relaxed_Initialization*" new feature of SPARK. At the early stages of releasing this feature, a related bug was captured and fixed by the benchmarks. Also, the SPARK *lemmas* documentation was added to *GNAT Studio*, as it significantly assisted the proving of some of the floating-point benchmarks via the use of lemmas. A list of further improvements has been documented within AdaCore that will be used to evolve the technology further.

6 Conclusion and Future Work

The hardening of existing code-bases is expected to become a commonplace security exercise due to the rise of industry mandated cyber-security standards and guidelines. This is especially predicted within aerospace with the recent adoption by EASA of the *ED-202A/ED-203A* security set as the currently only "Acceptable Means of Compliance" for aviation cyber-security airworthiness certification. The same is also true of the DO-326A set by the FAA. Under the new guidelines, security risk assessments will need to be performed on existing systems. It is expected that many of these activities will result in the identification of threat conditions, and in particular, the identification of situations where a security-related compromise of a system asset has a direct impact on safety. For each of these circumstances, it is not unreasonable to predict that a threat scenario analysis will identify the exposure of software vulnerabilities as a high "level of threat". This is particularly easy to argue where history has shown that an exposed vulnerability can be exploited for malicious intent. It may be that existing security measures mitigate the risk of the threat scenario occurring to an acceptable level; however, when this is not the case, additional measures need to be taken.

This is where SPARK can play an integral role when designing a new security architecture (or when security hardening an existing architecture). Identification of the security scope (the assets, perimeter and environment), performing a security risk assessment (identifying the threat conditions, threat scenarios, existing security measures) and performing a level of threat evaluation will ultimately lead to the identification of security-critical software components. With new systems, SPARK is the obvious choice for the development of critical security components. However, this report also shows that existing systems can benefit from the application of a SPARK hardening approach. Elevation of a security component, to SPARK silver level or higher, provides strong evidence to support a security effectiveness assurance argument; especially when arguing over the effectiveness of security measures against threat scenarios relating to application security vulnerabilities.

By design, SPARK aims to eradicate all security bugs, flaws, errors, faults,

holes, or weaknesses in software architecture regardless of if threat actors can exploit them. When proven to have achieved silver level or higher, the guaranteed AoRTE is a powerful countermeasure against cyber-attacks. This work demonstrated that the effort of adopting SPARK is not as hard as perceived since in an arguably short time, a relevant to the industry set of benchmarks from the EMBENCH benchmark suite were converted from C to SPARK and AoRTE was achieved in most of the cases, (see Table 2). If the complete functional specifications were available for the remaining, not fully proven benchmarks, the complete set of benchmarks could have achieved the silver level of SPARK. The SPARK technology was also able to identify a faulty benchmark, namely the *minver*, where its implementation was incomplete and producing the wrong results. Furthermore, as demonstrated in Section 5.1 the adoption of the SPARK technology did not significantly compromise the *C*-enabled performance. This and the significant security benefit of AoRTE make SPARK a default choice when it comes to the hardening of software libraries. Finally, in Section 4, we demonstrated the steps and best practices for adopting SPARK and highlighted the use of new features. These, and the provided references to external documentation, can be used as up-to-date guidelines for the easy adoption of the SPARK technology.

Future work will be focused on opportunities for hardening software libraries that are crucial for today's cyber-physical systems . Examples of such libraries can be network-protocols libraries and a high-assurance math libraries.

Acknowledgments

This research is part of the "High-Integrity Complex Large Software and Electronic Systems" (HICLASS) project that is supported by the Aerospace Technology Institute (ATI) Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture, under grant agreement No. 113213. The programme, delivered through a partnership between the ATI, Department for Business, Energy & Industrial Strategy (BEIS), and Innovate UK, addresses technology, capability, and supply chain challenges.

References

[1] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Chap-

ter – C Language Issues for Application Security. Pearson Education, 2006. ISBN: 9780132701938. URL: https://www.informit.com/articles/article.aspx?p=686170&seqNum=6. (accessed: 27.06.2020).

[2] Paul E. Black et al. *Source Code Security Analysis Tool Functional Specification Version 1.1*. NIST – Special Publication 500-268 v1.1. Feb. 2011. URL: https://www.nist.gov/publications/nist-sp-500-268-source-code-security-analysis-tool-function-specification-version-11. (accessed: 27.06.2020).

[3] *ISO/IEC TR 24772:2013 – Information technology – Programming languages – Guidance to avoiding vulnerabilities in programming languages through language selection and use*. 2013. URL: https://www.iso.org/standard/61457.html. (accessed: 24.06.2020).

[4] Peter C. Chapin and John W. McCormick. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. ISBN: 9781107040731.

[5] Philip Sparks. *The route to a trillion devices – The outlook for IoT investment to 2035*. ARM – White Paper. June 2017. URL: https://community.arm.com/iot/b/internet-of-things/posts/white-paper-the-route-to-a-trillion-devices. (accessed: 24.06.2020).

[6] AdaCore and Thales. *Implementation Guidance for the Adoption of SPARK*. 2018. URL: https://www.adacore.com/uploads/books/pdf/ePDF-ImplementationGuidanceSPARK.pdf.

[7] Rasim Alguliyev, Yadigar Imamverdiyev, and Lyudmila Sukhostat. "Cyber-physical systems and their security issues". In: *Computers in Industry* 100 (2018), pp. 212–223. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2018.04.017. URL: http://www.sciencedirect.com/science/article/pii/S0166361517304244.

[8] Roderick Chapman and Yannick Moy. *AdaCore Technologies for Cyber Security*. 2018. Chap. Appendix A – CWE Mapping, pp. 73–78. URL: https://www.adacore.com/uploads/books/pdf/AdaCore-Tech-Cyber-Security-web.pdf.

[9] "Ada: Meeting Tomorrow's Software Challenges Today - A White Paper by AdaCore". In: (2019). URL: https://www.adacore.com/uploads/techPapers/Ada-Meeting-Tomorrows-Software-Challenges-Today.pdf.

[10] Dor Zusman Ben Seri Gregory Vishnepolsky. *Urgent/11 – Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS*. ARMIS – White Paper. 2019. URL: https://info.armis.com/rs/645-PDC-047/images/Urgent11%5C%20Technical%5C%20White%5C%20Paper.pdf. (accessed: 24.06.2020).

[11] N. Neshenko et al. "Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations". In: *IEEE Communications Surveys Tutorials* 21.3 (2019), pp. 2702–2733.

[12] *ISO/IEC TR 24772-2:2020 – Programming languages – Guidance to avoiding vulnerabilities in programming languages – Part 2: Ada*. 2020. URL: https://www.iso.org/standard/71092.html. (accessed: 24.06.2020).

[13] *Ada 2012 Reference Manual - 4.6 Type Conversions*. URL: http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-4-6.html. (accessed: 17.06.2020).

[14] *Ada Expression Functions*. URL: https://docs.adacore.com/spark2014-docs/html/ug/en/source/specification_features.html#expression-functions. (accessed: 16.06.2020).

[15] AdaCore. *CodePeer: Comprehensive Static Analysis Toolsuite for Ada*. URL: https://www.adacore.com/codepeer/. (accessed: 12.06.2020).

[16] AdaCore. *Formal Verification with GNATprove*. URL: https://docs.adacore.com/spark2014-docs/html/ug/en/gnatprove.html. (accessed: 14.06.2020).

[17] AdaCore. *Intro To SPARK*. URL: https://learn.adacore.com/courses/intro-to-spark/index.html. (accessed: 14.06.2020).

[18] AdaCore. *SPARK 2014 Reference Manual*. URL: http://docs.adacore.com/spark2014-docs/html/lrm/. (accessed: 14.06.2020).

[19] AdaCore. *SPARK 2014 User's Guide*. URL: https://docs.adacore.com/spark2014-docs/html/ug/. (accessed: 14.06.2020).

[20] *American National Standards Institute*. URL: https://www.ansi.org/default. (accessed: 10.06.2020).

[21] *Aspect Relaxed_Initialization and Attribute Initialized*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/specification_features.html#aspect-relaxed-initialization-and-attribute-initialized. (accessed: 20.06.2020).

[22] *BEEBS Code Repository*. URL: https://github.com/mageec/beebs. (accessed: 10.06.2020).

[23] *BEEBS: Bristol/Embecosm Embedded Benchmark Suite*. URL: http://beebs.eu/. (accessed: 10.06.2020).

[24] Ben Brosgol. *When less is more: Programming language technology for safety*. URL: http://vita.mil-embedded.com/articles/when-programming-language-technology-safety/. (accessed: 12.06.2020).

[25] *CLOC – Count Lines of Code*. URL: http://cloc.sourceforge.net/. (accessed: 15.06.2020).

[26] EEMBC – Embedded Microprocessor Benchmark Consortium. *Dhrystone Benchmark*. URL: https://www.eembc.org/techlit/datasheets/ECLDhrystoneWhitePaper2.pdf. (accessed: 12.06.2020).

[27] *CoreMark – An EEMBC benchmark*. URL: https://www.eembc.org/coremark/. (accessed: 12.06.2020).

[28] The MITRE Corporation. *Common Weakness Enumeration (CWE) - A community-Developed List of Software & Hardware Weakness Types*. URL: http://cwe.mitre.org/index.html. (accessed: 12.06.2020).

[29] *Data Initialization Policy*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#data-initialization-policy. (accessed: 20.06.2020).

[30] *Discovery kit with STM32F407VG MCU*. URL: https://www.st.com/resource/en/user_manual/dm00039084-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf. (accessed: 21.06.2020).

[31] *EMBENCH aha_mont64 Benchmark*. URL: https://github.com/embench/embench-iot/blob/master/src/aha-mont64/mont64.c. (accessed: 19.06.2020).

[32] *EMBENCH user guide*. URL: https://github.com/embench/embench-iot/blob/master/doc/README.md. (accessed: 21.06.2020).

[33] *EMBENCH-IoT Code Repository*. URL: https://github.com/mageec/beebs. (accessed: 10.06.2020).

[34] Free and Open Source Silicon Foundation. *Embench: A Modern Embedded Benchmark Suite*. URL: https://embench.org/. (accessed: 10.06.2020).

[35] *GNAT User's Guide for Native Platforms*. URL: http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html. (accessed: 15.06.2020).

[36] *GNAT User's Guide Supplement for Cross Platforms*. URL: http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx.html. (accessed: 15.06.2020).

[37] *How to Investigate Unproved Checks*. URL: http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_investigate_unproved_checks.html. (accessed: 20.06.2020).

[38] *How to Run GNATprove*. URL: https://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_run_gnatprove.html. (accessed: 19.06.2020).

[39] *How to write loop invaritants*. URL: https://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_write_loop_invariants.html. (accessed: 19.06.2020).

[40] *Innovate UK*. URL: https://www.gov.uk/government/organisations/innovate-uk. (accessed: 10.06.2020).

[41] *International Organization for Standardization*. URL: https://www.iso.org/home.html. (accessed: 10.06.2020).

[42] *Introduction to Ada - Interfacing with C*. URL: https://learn.adacore.com/courses/intro-to-ada/chapters/interfacing_with_c.html. (accessed: 15.06.2020).

[43] *Introduction to SPARK - Global Contracts*. URL: https://learn.adacore.com/courses/intro-to-spark/chapters/02_Flow_Analysis.html?highlight=global%20aspect#global-contracts. (accessed: 17.06.2020).

[44] *MAGEEC: MAchine Guided Energy Efficient Compilation*. URL: http://mageec.org/. (accessed: 10.06.2020).

[45] *Manual Proof Using SPARK Lemma Library*. URL: https://docs.adacore.com/spark2014-docs/html/ug/gnatprove_by_example/manual_proof.html#manual-proof-using-spark-lemma-library. (accessed: 20.06.2020).

[46] Rick Merritt. *Embedded Benchmark Calls for Support*. URL: https://www.eetimes.com/embedded-benchmark-calls-for-support/#. (accessed: 10.06.2020).

[47] *Prof. David A. Patterson personal Webpage*. URL: https://www2.eecs.berkeley.edu/Faculty/Homepages/patterson.html. (accessed: 10.06.2020).

[48] *RISC-V: The Free and Open RISC Instruction Set Architecture*. URL: https://riscv.org/. (accessed: 10.06.2020).

[49] *SPARK 2014 – Analysis of Generics*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#analysis-of-generics. (accessed: 23.06.2020).

[50] *SPARK 2014 – Memory Ownership Policy*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#memory-ownership-policy. (accessed: 23.06.2020).

[51] *SPARK 2014 – Nonreturning Procedures*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#nonreturning-procedures. (accessed: 23.06.2020).

[52] *SPARK 2014 – Raising Exceptions and Other Error Signalling Mechanisms*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#raising-exceptions-and-other-error-signaling-mechanisms. (accessed: 23.06.2020).

[53] *SPARK 2014 Language Restrictions – Excluded Ada Features*. URL: http://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#excluded-ada-features. (accessed: 23.06.2020).

[54] *SPARK Lemma Library*. URL: https://docs.adacore.com/spark2014-docs/html/ug/en/source/spark_libraries.html#spark-lemma-library. (accessed: 20.06.2020).

[55] *The Arm Cortex-M4 Processor*. URL: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4. (accessed: 10.06.2020).