AdaCore Technologies for

CYBER SECURITY

Roderick Chapman & Yannick Moy

v 1.0

AdaCore Technologies for

Cyber Security

Roderick Chapman and Yannick Moy

Version 1.0

May 2018

About the Authors

Roderick Chapman

Dr. Roderick Chapman is an independent consulting engineer with more than 20 years of experience in the development and certification of critical software. Following graduation from the University of York in the U.K., Rod joined Praxis (now Altran UK), and contributed to many of the company's keynote projects, rising to the role of principal engineer for software process and design. He also led the programming language and verification research group at Praxis, leading the technical development, training, sales and marketing of the SPARK product line. Rod is a regular speaker at international conferences, and he is widely recognized as a leading authority on high integrity software development, programming language design, and software verification tools. In February 2015, Rod was appointed Honorary Visiting Professor in the Department of Computer Science at the University of York.

Yannick Moy

Dr. Yannick Moy is a Senior Software Engineer at AdaCore and co-director of the ProofInUse joint laboratory with INRIA (France). At AdaCore, he works on the software source code analyzers CodePeer and SPARK Pro, which are tools that are focused on detecting bugs or verifying safety/security properties. Yannick has led the development of the SPARK 2014 technology, which he has presented in articles, at conferences, in courses and in blogs (blog.adacore.com). Yannick previously worked on source code analyzers at PolySpace (now The MathWorks) and at the Université Paris-Sud.

Foreword

Building secure software is a challenging task. It seems that almost every week we read the news about yet another computer system that has failed in some way in the face of malicious or accidental misuse. "Cyber Security" is a wide-ranging field, spanning human factors, hardware design, sociology, and legal issues, in addition to software engineering. This book summarizes the contribution that the Ada and SPARK languages and AdaCore's tools can make to this final area—how to develop and verify correct and secure software.

Unlike AdaCore's previous guides to airborne and rail system software, this book does not follow the structure or requirements of a particular standard—in part because there is no widely used security standard that is required in practice. Instead it offers a more general treatment of the problem, but also includes an analysis of how AdaCore's technologies help address the weaknesses identified in the MITRE Corporation's Common Weakness Enumeration (CWE). The content is based on the authors' many years of practical experience in the development of high-end secure systems, the design of the Ada and SPARK programming languages, and research into program verification tools.

The book is intended for readers who are involved with software at any level (developers, project managers, procurement personnel) and who would like to learn how currently available technology can help address some of the most serious challenges associated with software and security. Our goal is to provide useful guidance both to those who are using other languages and are interested in the benefits that Ada offers, and to existing Ada users who might be confronted with new security requirements.

Please contact us if you have questions about any particular kind of software vulnerability, or how Ada, SPARK and their associated development and verification tools can help.

The authors gratefully acknowledge the assistance of Ben Brosgol (AdaCore) for his review of and contributions to the material presented in this document.

Roderick Chapman
Protean Code Limited
Bath, U.K.
May 2018

Yannick Moy
AdaCore
Paris, France
May 2018

info@adacore.com
www.adacore.com

AdaCore Technologies for Cyber Security

Contents

1. Introduction

It seems that every week's news brings a story about yet another high-profile failure of a computer system owing to a security issue. These problems have a significant impact on the public, businesses and government alike, affecting the reputation and share price of major organizations. In the most critical industry sectors, company directors face liability and governance concerns in an increasingly litigated environment.

In general, the "security" of computer systems can be characterized as a *weakest-link* scenario where attackers need only find and exploit a single weakness in any part of a *system*, including its hardware, software, operational environment, people, or operating procedures. Many attacks rely on social engineering, insiders, human factors, accidental misuse, and so on. These issues are important aspects of secure system design, but are not the focus of this document. Readers are referred to Ross Anderson's *Security Engineering* book [1], the US CERT website [2], or the recently established Cyber Security Body of Knowledge (CyBOK) project [3] for an overview of these wider issues.

Instead, this document focuses on a common attack vector—that of insufficiently secure software—and what can be done about it. More specifically, this book concentrates on the contribution that the Ada and SPARK languages and their associated tools can make. Ada was designed from the outset to support the needs of "high assurance" systems; its strengths in this area are now becoming more widely recognized as the operational environment has become more malicious, and hence less tolerant of defective software. From the outset, Ada was designed to emphasize readability, understanding, and verification. Many pernicious defects that plague other languages are absent from Ada. For example:

- Ada's syntax prevents several problems, including confusion of assignment and comparison, the "dangling else" problem, and the unintentional use of the null statement.
- Ada does not require the explicit use of "pointers" for low-level programming, parameter passing, or the use of array types. As

such, a huge number of "pointer-related" defects are entirely avoided.

- Ada's strong typing prevents a host of issues, including assignment of incompatible values to one another, confusion over "promotion" of types, and so on.
- Ada has high-level features for concurrent programming, freeing the programmer from low-level use of "locks" or semaphores and threads.

SPARK, a formally analyzable subset of Ada, inherits all of Ada's strengths and offers additional advantages for high-assurance software. These include the ability to mathematically prove program properties such as the correct uses of data, the absence of run-time errors, and even functional correctness with respect to a formally specified set of requirements.

Reader's Guide

Chapter 2 is recommended for all readers. It covers why producing secure software is such a challenge, and thus motivates our technical approach.

Chapter 3 covers the Ada and SPARK languages and then goes on to describe AdaCore's tools, with a focus on how they can support an evidence-based assurance case for security. Readers already familiar with Ada can probably skip section 3.1, while those familiar with AdaCore's tools as well can go straight on to chapter 4.

Chapter 4 presents a selection of common "vulnerabilities" in software, and describes how Ada and AdaCore's tools can address these issues. This selection is a very small subset of all such issues, but have been chosen since they present an opportunity to discuss areas where Ada and SPARK have a particular contribution to make.

Chapter 5 presents a number of industrial scenarios, from purely retrospective analysis of legacy systems, to new developments, to systems that involve mixed criticality and development technologies. These are intended as illustrations of real-world situations; suggestions for additional examples are welcome and will be considered for inclusion in future editions of this book.

Chapter 6 forms something of a “call to arms” for software developers.

Finally, two appendices cover the specifics of mapping Ada language and tool capabilities to the CWE enumeration, and a worked example of how the pernicious “SQL Injection” style of vulnerability can be handled with great panache in Ada and SPARK.

2. The Challenge of Secure Software

This chapter considers why building and operating secure computer systems appears to be so difficult, as evidenced by the frequency and magnitude of attacks reported in the media. Having set out the "bad news", this chapter closes with some principles that can be applied and justify AdaCore's position and technical approach.

2.1. Why is it so hard?

Security is a system-level property that is commonly defined as the protection of assets against threats that may compromise confidentiality, integrity, or availability. Thus, security means protection against unauthorized access to, corruption of, and denial of access to the assets. In a cyber system, software plays a key role in whether and how these requirements are met, and it does this in two ways:

- Providing the relevant security functionality (for example, cryptographic functions), and
- For the rest of the software, avoiding vulnerabilities that, if triggered, could violate the confidentiality, integrity and/or availability requirements.

In short, security entails demonstrating, with a level of confidence commensurate with the value of the assets, that the security functions do what they are supposed to do, and the rest of the system cannot do anything to place the assets at risk.

The development of secure software sets particular challenges that place it beyond a mere "quality control" problem, though. The following sub-sections expand on these challenges to set the scene and justify a rational technical approach.

The Malicious Environment

AdaCore customers have always been involved with building ultra-reliable *safety-critical* systems. In this world, the engineers usually have strict control of the software's operational environment, or the environment may be assumed to be *benign* or *well-behaved* in some way.

This is no longer the case. Software is often connected to open networks where benign behavior cannot be assumed. In short, software must be built to withstand and continue to operate correctly in an overtly malicious environment.

Ross Anderson and Roger Needham coined the phrase "Programming Satan's Computer" to characterize this challenge. Satan's computer doesn't fail *randomly* either—it fails *intelligently, in the worst-possible way, at the worst-possible time,* and it can fail in ways *that you don't even know about (yet…).*

Asymmetry of Capability

It gets worse. The "bad guys" (attackers, malicious actors) are smarter than you, have more money than you, and more time than you. Furthermore, their capabilities don't grow in some linear, predictable fashion. A public "leak" of a government's arsenal of hacking tools can put nation-state level capability within reach of anyone at all in a matter of days, and these capabilities become commoditized very rapidly.

Asymmetry of Effort

The software developer is responsible for preventing every possible defect that might lead to any sort of security exploit or failure of the system. An attacker, on the other hand, has to find *just one* defect to mount a successful attack.

Asymmetry of Knowledge

Computer security (and, in particular, its sub-genre *cryptography*) is a strange discipline, since the total knowledge of the field is far greater than what is published in the scientific literature. In short, various groups (including "attackers", and certain notable government agencies) know things that typical developers don't. Examples include the RSA cryptosystem in the early 1970's, Differential Power Analysis pre-1999, and (until very recently) the so-called “Spectre” and “Meltdown” problems in modern CPUs.

This sets the software developer with a particular challenge: to defend software against attacks that the developer doesn’t even know about.

Asymmetry of Impact

In software, it is very difficult to predict the relationship between a particular defect (or class of defects) and its potential impact on the system’s behavior, the system’s customers/users, or the developing organization’s business. The impact of security issues can range from negligible to those that destroy the reputation and share-price of a company overnight. Given that attacks exist that developers don’t know about, trying to decide whether a particular defect is “high impact” (and therefore worthy of being fixed) can be almost impossible.

The Limits of Test

In the 1972 Turing Award lecture, Edsger Dijkstra famously pointed out that “program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.”

Given an arbitrarily intelligent and motivated attacker, developers of secure systems must assume that systems will be attacked with input data and in states *that have never been tested*. For any non-trivial software system, any claim of security solely based on “lots of testing” must be regarded with extreme skepticism. Even a well-organized and independent Penetration Testing activity offers no guarantee of finding “all the bugs”.

The Limits of Talent

Some projects like to claim that their software quality is OK “because we only hire really good people...” or something like that. Such claims do not stand up to the most cursory inspection. Data from the Personal Software Process / Team Software Process (PSP/TSP) group at the US Software Engineering Institute show that even the best performing programmers inject around 20 defects per 1000 logical lines of code in their work. Truly critical systems might aim for 0.2 defect per kloc delivered to the customer—a factor of 100 times better, or equivalent to those programmers preventing or finding and fixing 99 out of every 100 defects.

It is also obvious to note that this approach does not scale with team size—projects need technologies and disciplines that allow *all* developers to produce work of the required quality, instead of relying on a few "hero programmers" to save the day (or the project, or the company…)

The "First Release" Problem

There is lots of advice that tells us to "patch" our software regularly to make sure we've got the latest (and presumably least-buggy) version. This is reasonable advice, but can be something of a challenge to implement with appropriate levels of authentication and confidentiality, particularly for small embedded devices.

For a developer, the ability to "patch" should not be an excuse to ship defective software to the customer, thinking that it can be patched later if defects are reported. The problem is that some defects are of such dramatic impact, and the damage is done so fast, that there is no time or opportunity for any corrective action. As an example, consider the flight control software for an aerodynamically unstable fighter aircraft. A developer approaching the test pilot before first flight and saying "Don't worry about the bugs, we'll ship you a patch…" would soon be looking for a new career.

While "improve and patch" is a reasonable model for many software vendors, some software just has to be *fit-for-purpose* at the point of its *first release*. Additionally, a demonstration of fitness for purpose might need to be submitted to an independent authority or regulator. This problem requires a fundamental shift in engineering mind-set: one that is embodied in Ada itself and many of AdaCore's key technologies.

2.2. Standards and Guidance

"The nice thing about standards is that you have so many to choose from."
– Andrew S Tanenbaum

Given the perceived scale and threat from “Cyber Security”, it is no surprise that a great deal has been written about the problem, and what to do about it.

The European Cyber Security Organisation has published an “Overview of existing Cybersecurity standards and certification schemes” [4]. The document (which is just a survey) is nearly 200 pages, and lists something like 107 different guidance documents and/or standards for system development, and goes on to cite nine schemes for certification of professional skills.

Even within a particular application domain or industry sector, there can still be an overwhelming volume of guidance and “standards” that appear to apply. Standards also vary widely in terms of the vigor with which they are enforced (from effectively mandatory to entirely optional) and their technical demands (from highly specific and onerous to general and lax).

There have also been some higher-level attempts to look at the problem which merit some attention. Examples include the US National Academy of Sciences report “Software for Dependable Systems: Sufficient Evidence?” [5] and the US National Institute for Standards and Technology (NIST) report “Dramatically Reducing Software Vulnerabilities” [6]. The NIST report identifies five approaches that have the potential to make a dramatic impact on software quality, including the use of formal methods, which (as will be shown later) aligns closely with AdaCore’s capabilities and technologies.

2.3. The Market, Lemons, and Regulators

For many years, there was hope that “the market” would self-correct and produce a rational approach to the development of secure software. This has not happened. Firstly, the development of “secure” software has been seen as something of a specialist “niche”, but development disciplines and approaches have been dominated by the larger market for “not so secure” software where time-to-market and “features” take precedence over quality. Secondly, for a buyer of software it is almost impossible to tell if some specific product is any more or less secure than any other product, since proprietary software is often cloaked in secrecy (“security by obscurity”) and restricted by prohibitive licensing terms. Economists call this “A market for Lemons” after similar observations were made about the market for used cars.

In a significant change of stance, the UK’s National Cyber Security Strategy 2016-2021 (NCSS) explicitly notes:

> “But the combination of market forces and government encouragement has not been sufficient in itself to secure our long-term interests in cyberspace at the pace required. Too many networks, including in critical sectors, are still insecure. The market is not valuing, and therefore not managing, cyber risk correctly.” [7 section 4.13]

The NCSS goes on to promise more active “intervention” in critical areas. It is not yet clear what form these interventions will take, but the challenge is clear: system and software developers must be ready to improve, justify and defend their practices before their national regulator (or worse, their insurer or a court of law) decides to step in.

2.4. A Manifesto for Secure Software

Although this picture may seem bleak, progress is possible.

AdaCore is primarily involved with the design of software development and verification tools, but these are not the sole route to improvement. Great engineering involves an interplay between technologies (tool, languages etc.), people (their skills, disciplines, attitudes), and engineering processes.

These three elements impact one another—for example, the introduction of a new static verification tool on a project might mean that later code review processes can be modified, and that engineers change their disciplines and “coding style” as they learn how to get best results from the tool.

This section lays out some basic principles for high-integrity software engineering that could be applied to any project, regardless of the standards, industry, or regulatory environment that might apply.
Chapter 3 goes into more details, showing how AdaCore’s languages and tool technologies contribute to meeting these goals.

Requirements

Arguably the most important aspect of producing secure and reliable software, and also perhaps the most difficult to achieve, is to specify a complete, consistent, and unambiguous set of requirements that the software must meet, and to do so without overly constraining the solution. The requirements should specify the "what", but not the "how", and should be expressed in a way that facilitates verifying whether they are met. Software standards such as DO-178C (airborne software) and CENELEC EN 50128 (railway control and protection systems) recognize the critical role of the requirements specification in the software life cycle. Typically derived from overall system requirements, software requirements are most easily visualized as comprising several tiers:

- High-level requirements that relate to overall functionality, performance, capacity /response time, interface issues, usability, and safety and/or security. These requirements drive the design of the software architecture.
- Low-level requirements that emerge from the software design. These requirements are defined for each component and in particular establish what each code module (subprogram, in Ada parlance) assumes when it is invoked and what it promises to deliver when it returns.

A language such as Ada or SPARK can serve to specify requirements at both levels, in particular through the various forms of contracts (pre- and postconditions, type invariants). The developer can thus directly embed requirements in the source code and verify compliance either statically with SPARK proofs or dynamically with Ada run-time checks. SPARK also has the advantage of an unambiguous notation that can formalize requirements such as the information flow between components, key integrity properties for security and/or safety, and the detailed semantics of what a subprogram computes.

Security requirements should be formulated at the outset; issues such as the usage environment (standalone with trusted operations personnel versus networked and accessible from unvetted parties) have an obvious effect on the design and the relevant assurance level. The security-related interactions between the system and its environment need to be defined, the threats identified, and countermeasures specified. (An example of how security-related issues are being assessed in the commercial avionics industry may be found in DO-326, Airworthiness Security Process Specification.)

Security-related requirements established at the system level flow down through the software life cycle into high-level requirements (for example the strength of a cryptography function) and ultimately into low-level requirements and then source code. The chosen programming language(s) and tools have a significant effect on the ease or difficulty of demonstrating that the resulting code in fact correctly implements the requirements. As will be explained throughout this book, the Ada and SPARK languages together with AdaCore's development and verification tools offer particular advantages.

Architecture

A system's architecture entails its high-level design as components with well-defined interfaces and interrelationships. A good architecture provides a solid framework with effective modularization and robustness in the presence of future enhancements and requirements changes. Architecture covers issues such as redundancy, the provision of fail-safe states or modes, mitigation of security concerns by physical means (e.g. hardware design), and the *separation* of critical from non-critical components. The last of these also allows the most critical software components to be *as small as possible*, which helps to control cost.

At a more technical level, strong architecture and separation means that the most appropriate languages, tools and technologies can be used where they are best suited. For example, in the MULTOS CA system, developed by Praxis in the UK [8], the security kernel of the system was implemented and verified using the SPARK toolset, while the GUI was implemented in C++, based on a deliberate and strict separation of security concerns.

Evidence-based Assurance

The fitness-for-purpose of a system should be justified by a *logical argument* which is supported by *evidence* from a wide range of sources. The evidence might be based on analysis of design artefacts, observation including all forms of testing, metrics, and both direct and indirect evidence of process compliance.

Verification-Driven Development

Given a need for evidence, a development approach should be chosen that generates the evidence as a natural by-product. This can be summarized as “Security should be built in, not bolted on.” This idea can be seen as a generalization of “Test-Driven Development” to cover *all* the forms of verification activity that are available.

In line with the Agile manifesto, verification tasks should be *automated* as far as possible, and embodied in a continuous integration pipeline.

Analysis over Observation

“Talk is Cheap...Show me the Code.”
– Linus Torvalds

Verification activities can be categorized as “static” or “dynamic.”

Static Verification (also known as “Static Analysis”) concentrates on *analysis* of development artefacts (e.g. designs, models, source code) without actually running the system. Static analyses can be performed by humans (for example, personal and peer review), or automated by machines (for example, use of a tool like CodePeer or SPARK Pro.)

Dynamic Verification constitutes all activities that involve *observation* of the system, in either a simulated or real-world environment, so covers all forms of “testing.”

As noted earlier, testing of security-critical systems has severe limitations. In short, the results are only as good as the test data that have been exercised, and that will always a tiny fraction of the possible system states and inputs. Further, it must be assumed that attackers will find “tests” that the system’s developers haven’t tried.

In theory, a static verification activity should yield results that are true for *all possible states and inputs*, and therefore offers a qualitatively different level of assurance from testing alone. This tool property is referred to as “Soundness”, meaning that the results of a static verification really are trustworthy. The concept is best illustrated with a simple example. Imagine submitting a program’s source code to a tool, asking the question “Are there any defects?” (for example, reads of uninitialized variables) and receiving a report in response. If the tool is Sound, then these are the *only* such defects; if the tool is Unsound then the code may have defects that were not reported. Phrased differently: if a Sound tool reports that the program has no defects, then this conclusion can be trusted. If an Unsound tool reports that the program has no defects, then it is still possible that unreported defects are present.

The Sound tool offers a more desirable result than the Unsound tool, since it provides a higher degree of confidence and requires less testing and re-work later.

For these reasons, a verification approach that emphasizes the use of *sound, automated* and *static* verification is recommended.

But…as always, there is no “free lunch.” It turns out that Soundness in static verification is rather difficult to achieve, for the reasons set out in the next sub-section. And all tools, whether Sound or Unsound, run the risk of generating “false alarms”: reporting a potential defect that in fact is not a problem. An ideal tool that is both Sound (reporting all defects) and precise (not reporting any non-defects) is not achievable; in practice a trade-off must be maintained. For example, first applying an Unsound but precise tool to analyze legacy code and correct the reported defects, and then using a Sound tool to identify the remaining defects.

Unambiguous Notation

To analyze the meaning (and therefore the presence or absence of important defects) of a program successfully, a tool needs to know *exactly* what a program means. This seems obvious, but turns out to be rather difficult to achieve with most of the popular and practical programming languages.

The problem is *ambiguity* in programming languages—there are features or scenarios where the meaning of a program is said to be *undefined* or *unspecified* by the definition of the language. This is easy for a compiler to resolve—it just chooses one of a range of options and carries on—but is something of a disaster for static verification tools. If a verification tool has to “guess” the meaning of an undefined behavior in a program, then the results can be Unsound.

AdaCore's approach is to achieve Soundness in static verification as far as is possible, and with respect to a reasonable and practical set of assumptions.

In the interest of run-time performance the Ada language has provision for unspecified behavior and compiler-dependent choices. For example, the order of evaluation of sub-expressions inside an expression is unspecified, which can lead to different results depending on the order chosen if these sub-expressions have side-effects. In particular, the association of additions in an expression A + B + C is compiler dependent, which can lead to different results depending on the order chosen if one of these choices leads to an integer overflow. For each of these, the CodePeer static analyzer makes and documents its choices, so that they are as close as possible to the choices made in general, and in particular in the GNAT compiler. But it cannot guarantee that these coincide with the choices made by a specific compiler version with a specific host and target configuration, so these choices should be inspected if they could be relevant for the static analysis.

The SPARK language and toolset set a high-water mark in this regard. Soundness is achieved through a combination of language subsetting, additional language rules, and analyses. SPARK can also be thought of as a fully *Formal* language owing to its unambiguous semantics.

3. Languages, Tools and Technologies Overview

This chapter summarizes the Ada and SPARK languages, as well as AdaCore's tools and technologies, and highlights their contributions to system security.

3.1. Ada

Ada is a modern programming language designed for large, long-lived applications – and embedded systems in particular – where reliability, maintainability, and efficiency are essential. It was originally developed in the early 1980s (this version is generally known as Ada 83) by a team led by Jean Ichbiah at CII-Honeywell-Bull in France. The language was revised and enhanced in an upward-compatible fashion in the early 1990s, under the leadership of Tucker Taft from Intermetrics in the U.S. The resulting language, Ada 95, was the first internationally standardized (ISO) object-oriented language. Under the auspices of ISO, a further (minor) revision was completed as an amendment to the standard; this version of the language is known as Ada 2005. Additional features (including support for contract-based programming in the form of subprogram pre- and postconditions and type invariants) were added in the most recent version of the language standard, Ada 2012 (see [9,10,11] for information about Ada).

The name "Ada" is not an acronym; it was chosen in honor of Augusta Ada Lovelace (1815-1852), a mathematician who is sometimes regarded as the world's first programmer because of her work with Charles Babbage. She was also the daughter of the poet Lord Byron.

Ada is seeing significant usage worldwide in high-integrity / safety-critical / high-security domains including commercial and military aircraft avionics, air traffic control, space applications, railroad systems, and medical devices. With its embodiment of modern software engineering principles Ada is an excellent teaching language for both introductory and advanced computer science courses, and it has been the subject of significant university research especially in the area of real-time technologies.

AdaCore has a long history and close connection with the Ada programming language. Company members worked on the original Ada 83 design and review and played key roles in the Ada 95 project as well as the subsequent revisions. The initial GNAT compiler was essential to the growth of Ada 95; it was delivered at the time of the language's standardization, thus guaranteeing that users would have a quality implementation for transitioning to Ada 95 from Ada 83 or other languages.

Language Overview

Ada is multi-faceted. From one perspective it is a classical stack-based general-purpose language, not tied to any specific development methodology. It has a simple syntax, structured control statements, flexible data composition facilities, strong type checking, traditional features for code modularization ("subprograms"), and a mechanism for detecting and responding to exceptional run-time conditions ("exception handling").

But it also includes much more:

Scalar ranges

Unlike languages based on C (such as C++, Java, and C#), Ada allows the programmer to simply and explicitly specify the range of values that are permitted for variables of scalar types (integer, floating-point, fixed-point, and enumeration types). The attempted assignment of an out-of-range value causes a run-time error. The ability to specify range constraints makes programmer intent explicit and makes it easier to detect a major source of coding and user input errors. It also provides useful information to static analysis tools and facilitates automated proofs of program properties.

Here's an example of an integer scalar range:

```
Score : Integer range 1..100;
N     : Integer;
...
Score := N;
-- A run-time check verifies that N is within the range 1..100
-- If this check fails, a Constraint_Error exception is raised
```

Contract-based programming

Ada 2012 allows extending a subprogram specification or a type/subtype declaration with a contract (a Boolean assertion). Subprogram contracts take the form of *preconditions* and *postconditions*. Through contracts the developer can formalize the intended behavior of the application, and can verify this behavior by testing, static analysis or formal proof.

Here's a skeletal example that illustrates contact-based programming; a `Table` object is a fixed-length container for distinct `Float` values.

```
package Table_Pkg is
   type Table is private;  -- Encapsulated type

   function Is_Full  (T    : in Table) return Boolean;
   function Contains (T    : in Table;
                      Item : in Float) return Boolean;

   procedure Insert (T : in out Table; Item: in Float)
     with Pre  => not Is_Full(T) and not Contains(T, Item),
          Post => Contains(T, Item);

   procedure Remove (T : in out Table; Item: out Float);
     with Pre  => Contains(T, Item),
          Post => not Contains(T, Item);
   ...
private
   ... -- Full declaration of Table
end Table_Pkg;
```

A compiler option controls whether the pre- and post-conditions are checked at run time. If checks are enabled, a failure raises the `Assertion_Error` exception.

Ada 2012 goes further still, allowing type invariants and subtype predicates to specify *precisely* what is and isn't valid for any particular (sub)type, including composite types such as records and arrays. For example, one can easily specify that field `Max_A` in the `Launching_Pad` structure below is the maximal value of angle allowed given the distance D to the center of the launching pad and the height H of the rocket, with the guarantee that automatic run-time checks will be inserted by the compiler to verify this predicate as well as constraints on the individual fields:

```ada
type Launching_Pad is record
   D, H   : Length;
   Max_A : Angle;
end record
  with Predicate => Angle (Arctan (H, D)) <= Max_A;
```

Programming in the large

The original Ada 83 design introduced the package construct, a feature that supports encapsulation ("information hiding") and modularization, and which allows the developer to control the namespace that is accessible within a given compilation unit. Ada 95 introduced the concept of "child units," adding considerable flexibility and easing the design of very large systems. Ada 2005 extended the language's modularization facilities by allowing mutual references between package specifications, thus making it easier to interface with languages such as Java.

Generic templates

A key to reusable components is a mechanism for parameterizing modules with respect to data types and other program entities, for example a stack package for an arbitrary element type. Ada meets this requirement through a facility known as "generics"; since the parameterization is done at compile time, run-time performance is not penalized.

Object-Oriented Programming (OOP)

Ada 83 was object-based, allowing the partitioning of a system into modules (packages) corresponding to abstract data types or abstract objects. Full OOP support was not provided since, first, it seemed not to be required in the real-time domain that was Ada's primary target, and, second, the apparent need for automatic garbage collection in an OO language would have interfered with predictable and efficient performance.

However, large real-time systems often have components such as GUIs that do not have real-time constraints and that could be most effectively developed using OOP features. In part for this reason, Ada 95 added comprehensive support for OOP, through its "tagged type" facility: classes, polymorphism, inheritance, and dynamic binding. These features do not require automatic garbage collection; instead, definitional features introduced by Ada 95 allow the developer to supply type-specific storage reclamation operations ("finalization"). Ada 2005 brought additional OOP features including Java-like interfaces and traditional `obj.op(...)` operation invocation notation.

Ada is methodologically neutral and does not impose a "distributed overhead" for OOP. If an application does not need OOP, then the OOP features do not have to be used, and there is no run-time penalty.

See [11] or [12] for more details.

Concurrent programming

Ada supplies a structured, high-level facility for concurrency. The unit of concurrency is a program entity known as a "task." Tasks can communicate implicitly via shared data or explicitly via a synchronous control mechanism known as the rendezvous. A shared data item can be defined abstractly as a "protected object" (a feature introduced in Ada 95), with operations executed under mutual exclusion when invoked from multiple tasks. Asynchronous task interactions are also supported, specifically timeouts and task termination. Such asynchronous behavior is deferred during certain operations, to prevent the possibility of leaving shared data in an inconsistent state. Mechanisms designed to help take advantage of multi-core architectures were introduced in Ada 2012.

Systems programming

Both in the “core” language and the Systems Programming Annex, Ada supplies the necessary features for hardware-specific processing. For example, the programmer can specify the bit layout for fields in a record, define alignment and size properties, place data at specific machine addresses, and express specialized code sequences in assembly language. Interrupt handlers can also be written in Ada, using the protected type facility.

Real-time programming

Ada’s tasking facility and the Real-Time Systems Annex support common idioms such as periodic or event-driven tasks, with features that can help avoid unbounded priority inversions. A protected object locking policy is defined that uses priority ceilings; this has an especially efficient implementation in Ada (mutexes are not required) since protected operations are not allowed to block. Ada 95 defined a task dispatching policy that basically requires tasks to run until blocked or preempted, and Ada 2005 introduced several others including Earliest Deadline First.

High-integrity systems

With its emphasis on sound software engineering principles Ada supports the development of high-integrity applications, including those that need to be certified against safety standards such DO-178C for avionics, CENELEC EN 50128 for rail systems and security standards such as the Common Criteria. For example, strong typing means that data intended for one purpose will not be accessed via inappropriate operations; errors such as treating pointers as integers (or vice versa) are prevented. And Ada’s array bounds checking prevents buffer overflow vulnerabilities that are common in C and C++.

However, the full language may be inappropriate in a safety- or security-critical application, since the generality and flexibility of some features – especially those with complex run-time semantics – complicates analysis and could interfere with traceability / certification requirements. Ada addresses this issue by supplying a compiler directive, `pragma Restrictions`, that allows constraining the language features to a well-defined subset (for example, excluding dynamic OOP facilities).

The evolution of Ada has seen the continued increase in support for safety-critical and high-security applications. Ada 2005 standardized the Ravenscar Profile, a collection of concurrency features that are powerful enough for real-time programming but simple enough to make certification practical. Ada 2012 has introduced contract-based programming facilities, allowing the programmer to specify preconditions and/or postconditions for subprograms, and invariants for encapsulated (private) types. These can serve both for run-time checking and as input to static analysis tools.

In brief, Ada is an internationally standardized language combining object-oriented programming features, well-engineered concurrency facilities, real-time support, and built-in reliability through both compile-time and run-time checks. As such it is an appropriate language for addressing the real issues facing software developers today. Ada is used throughout a number of major industries to design software that protects businesses and lives.

3.2. SPARK

SPARK[1] is a software development technology (programming language and verification toolset) specifically designed for engineering ultra-low defect level applications, for example where safety and/or security are key requirements. SPARK Pro is the commercial-grade offering of the SPARK technology developed by AdaCore and Altran. The main component in the toolset is GNATprove, which performs formal verification on SPARK code.

SPARK has an extensive industrial track record. Since its inception in the late 1980s it has been used worldwide in a range of industrial applications such as civil and military avionics, air traffic management / control, railway signaling, cryptographic software, and cross-domain solutions. SPARK 2014 is the most recent version of the technology (see [13]).

[1] Note that our SPARK is totally unrelated to the Apache SPARK analytics framework, or the SPARC CPU Instruction Set Architecture.

Formality and Soundness

Two major design goals of SPARK are the provision of an *unambiguous* and *formal* semantics, which therefore permits the *soundness* of static verification. These goals have always been at the heart of SPARK's design. Soundness builds trust in the tools, supports evidence-based assurance, completely removes many classes of dangerous defects, and allows subsequent verification activities (e.g. testing) to be cheaper (owing to less rework) or eliminated entirely.

Flexibility

SPARK 2014 offers the flexibility of configuring the language on a per-project basis. Restrictions can be fine-tuned based on the relevant coding standards or run-time environments.

SPARK 2014 code can easily be combined with full Ada code or with C, so that new systems can be built on and reuse legacy codebases.

Ease of Adoption

The SPARK 2014 technology is easy to learn and can be smoothly integrated into an organization's existing development and verification methodology and infrastructure.

Pre-2014 versions of the SPARK language used a special annotation syntax for the various forms of contracts. In SPARK 2014 this has been merged with the standard Ada 2012 contract syntax, which both simplifies the learning process and also allows new paradigms of software verification. Programmers familiar with writing executable contracts for run-time assertion checking can use the same approach but with additional flexibility: the contracts can be verified either dynamically through classical run-time testing methods or statically (i.e., pre-compilation and pre-test) using automated tools.

SPARK supports "hybrid verification" that can mix testing with formal proofs. For example an existing project in Ada and C can adopt SPARK to implement new functionality for critical components. The SPARK units can be analyzed statically to achieve the desired level of verification, with testing performed at the interfaces between the SPARK units and the modules in the other languages.

Reduced Test Effort and Cost

Software verification typically involves extensive testing, including unit tests and integration tests. Traditional testing methodologies are a major contributor to the high delivery costs for safety-critical software. Furthermore, they may fail to detect errors. SPARK 2014 addresses this issue by allowing automated proof to be used to demonstrate functional correctness at the subprogram level, either in combination with or as a replacement for unit testing. In the high proportion of cases where proofs can be discharged automatically the cost of writing unit tests is completely avoided. Moreover, verification by proofs covers all execution conditions and not just a sample.

3.3. GNAT Pro Enterprise

GNAT Pro Enterprise is an Ada and C development environment for producing critical software systems where reliability, efficiency and maintainability are essential.

Based on the GNU GCC technology, GNAT Pro Enterprise supports all versions of the Ada language standard, from Ada 83 to Ada 2012, and also handles multiple versions of C (C89, C99, and C11). It includes an Integrated Development Environment (GNAT Programming Studio and/or GNATbench), a comprehensive toolsuite including a visual debugger, and a set of libraries and bindings.

GNAT Pro Enterprise offers several features that make it ideal for the development of secure systems. These include:

Configurable Run-Time Library

GNAT Pro Enterprise includes a configurable run-time capability, which allows specifying support for Ada's dynamic features in an *a la carte* fashion ranging from none at all to full Ada. The units included in the executable may be either a subset of the standard libraries provided with GNAT Pro, or specially tailored to the application. For the most critical applications and "bare metal" systems, the Zero FootPrint (ZFP) run time offers a truly minimal application footprint (rivalling that of C) while retaining compatibility with the SPARK subset and verification tools.

Full Ada 83 to 2012 Implementation

GNAT Pro provides a complete implementation of all versions of the Ada language standard, from Ada 83 to Ada 2012. Developers of safety-critical and high-security systems can thus take advantage of features such as contract-based programming.

Enhanced Data Validity Checking

Improper or absent data validity checking is a pernicious source of security vulnerabilities in software systems. Ada has always offered range checks for scalar subtypes, but GNAT Pro goes further, offering enhanced validity checking that can protect a program against malicious or accidental memory corruption, failed I/O devices, and so on. This feature is particularly useful in combination with automatic *Fuzz Testing*, since its offers strong defense for invalid data *at the software boundary* of a system.

Support and Expertise

At the heart of every AdaCore subscription are the support services that AdaCore provides to its customers. AdaCore staff are recognized experts on the Ada language, software certification standards in several domains, compilation technologies, and static and dynamic verification. They have extensive experience in supporting customers in avionics, railway, space, energy, air traffic management/control, and military projects.

Every AdaCore product comes with front-line support provided directly by these experts, who are also the developers of the technology. This ensures that customers' questions (requests for guidance on feature usage, suggestions for technology enhancements, or defect reports) are handled efficiently and effectively.

Beyond this bundled support, AdaCore also provides Ada language and tool training as well as on-site consulting on topics such as how to best deploy the technology, and assistance on start-up issues. On-demand tool development or ports to new platforms are also available.

3.4. GNAT Pro Assurance

GNAT Pro Assurance adds specialized support, such as bug fixes and "known problems" analyses, on a specific version of the toolchain. This product line is especially suitable for applications with long-lived maintenance cycles or assurance requirements, since critical updates to the compiler or other product components may become necessary years after the initial release.

Sustained Branches

Unique to GNAT Pro Assurance is a service known as a “sustained branch”: customized support and maintenance for a specific version of the product. A project on a sustained branch can monitor relevant known problems, analyze their impact, and if needed update to a newer version of the product on the same development branch (i.e., not incorporating changes introduced in later versions of the product).

Sustained branches are a practical solution to the problem of ensuring toolchain stability while allowing flexibility in case an upgrade is needed to correct a critical problem.

Source to Object Traceability

A compiler option can limit the use of language constructs that generate object code that is not directly traceable to the source code. As an add-on service, AdaCore can perform an analysis that demonstrates this traceability and justifies any remaining cases of non-traceable code.

3.5. Static Verification - Basic Tools

GNATmetric

The GNATmetric tool analyzes source code to calculate a set of commonly used industry metrics, thus allowing developers to estimate the size and better understand the structure of the source code. This information also facilitates satisfying the requirements of certain software development frameworks.

GNATcheck

GNATcheck is a coding standard verification tool that is extensible and rule-based. It allows developers to completely define a coding standard as a set of rules, for example a subset of permitted language features. It verifies a program’s conformance with the resulting rules and thereby facilitates demonstration of a system’s compliance with certification standards.

Key features include:

- An integrated Ada Restrictions mechanism for banning specific features from an application. This can be used to restrict features such as tasking, exceptions, dynamic allocation, fixed- or floating point, input/output and unchecked conversions.

- Restrictions specific to GNAT Pro, such as banning features that result in the generation of implicit loops or conditionals in the object code, or in the generation of elaboration code.

- Additional Ada semantic rules resulting from customer input, such as ordering of parameters, normalized naming of entities, and subprograms with multiple returns.

- Easy-to-use interface for creating and using a complete coding standard.

- Generation of project-wide reports, including evidence of the level of compliance with a given coding standard.

- Over 30 compile-time warnings from GNAT Pro that detect typical error situations, such as local variables being used before being initialized, incorrect assumptions about array lower bounds, infinite recursion, incorrect data alignment, and accidental hiding of names.

- Style checks that allow developers to control indentation, casing, comment style, and nesting level.

GNATstack

GNATstack is a software analysis tool that enables Ada/C software development teams to accurately predict the maximum size of the memory stack required for program execution.

The GNATstack tool statically computes the maximum stack space required by each task in an application. The reported bounds can be used to ensure that sufficient space is reserved, thus guaranteeing safe execution with respect to stack usage. The tool uses a conservative analysis (and user-supplied input) to deal with complexities such as subprogram recursion, while avoiding unnecessarily pessimistic estimates.

GNATstack exploits data generated by the compiler to compute worst-case stack requirements. It performs per-subprogram stack usage computation combined with control flow analysis.

GNATstack can analyze object-oriented applications, automatically determining maximum stack usage on code that uses dynamic dispatching in Ada. A dispatching call challenges static analysis because the identity of the subprogram being invoked is not known until run time. GNATstack solves this problem by statically determining the subset of potential targets (primitive operations) for every dispatching call. This significantly reduces the analysis effort and yields precise stack usage bounds on complex Ada code.

This is a static analysis tool in the sense that its computation is based on information known at compile time. When the tool indicates that the result is accurate, the computed bound can never be exceeded.

On the other hand, there may be cases in which the results will not be accurate (the tool will report such situations) because of some missing information (such as the maximum depth of subprogram recursion, indirect calls, etc.). The user can assist the tool by specifying missing call graph and stack usage information.

GNATstack's main output is the worst-case stack usage for every entry point, together with the paths that result in these stack sizes. The list of entry points can be automatically computed (all the tasks, including the environment task) or can be specified by the user (a list of entry points or all the subprograms matching a given regular expression).

GNATstack can also detect and display a list of potential problems when computing stack requirements:

- Indirect (including dispatching) calls. The tool will indicate the number of indirect calls made from any subprogram.

- External calls. The tool displays all the subprograms that are reachable from any entry point for which there is no stack or call graph information.

- Unbounded frames. The tool displays all the subprograms that are reachable from any entry point with an unbounded stack requirement.

The required stack size depends on the arguments passed to the subprogram. For example:

```
procedure P (N : Integer) is
   S : String (1 .. N);
begin
   ...
end P;
```

- Cycles. The tool can detect all the cycles (i.e., potential recursion) in the call graph.

GNATstack allows the user to supply a text file with the missing information, such as the potential targets for indirect calls, the stack requirements for externals calls, and the maximal size for unbounded frames.

Timing Verification

Suitably subsetted, Ada (and SPARK) are also amenable to the static analysis of timing behavior. This kind of analysis is relevant for real-time systems, where *worst-case execution time (WCET)* must be known in order to guarantee that timing deadlines will always be met. Timing analysis is also of interest for secure systems, where the issue might be to show that programs do not leak information via so-called *side-channels* based on the observation of *differences* in execution time.

AdaCore does not produce its own WCET tool, but there are several such tools on the market from partner companies, such as RapiTime from Rapita Systems Ltd.

Memory Usage Verification

Ada and SPARK can support the static analysis of worst-case memory consumption, so that a developer can show that a program will *never* run out of memory at execution time.

SPARK can be compiled with no heap data structure at run time, so memory usage analysis reduces to a worst-case analysis of stack usage for each task in a system. This is implemented directly in AdaCore's GNATstack tool, as described above.

Semantic Analysis Tools—Libadalang

Libadalang is a reusable library that forms a high-performance semantic processing and transformation engine for Ada source code. In some ways it is similar to ASIS (see below), but exposes its API in Java and Python as well as Ada. It is particularly suitable for writing *lightweight* and *project-specific* static analysis tools.

An example of a potential Libadalang application is the enforcement of a particular naming convention—perhaps a rule for the naming of types that contain security-critical data. This is outside the scope of general-purpose tools like GNATcheck or CodePeer but is simple to express in Libadalang.

Semantic Analysis Tools—ASIS and GNAT2XML

ASIS, the Ada Semantic Interface Specification, is a library that gives applications access to the complete syntactic and semantic structure of an Ada compilation unit. This library is typically used by tools that need to perform some sort of static analysis on an Ada program.

ASIS is an international standard (ISO/IEC 15291:1995) and is designed to be compiler independent. Thus, a tool that processes the ASIS representation of a program will work regardless of which ASIS implementation has been used. ASIS-for-GNAT is AdaCore's implementation of the ASIS standard, for use with the GNAT Pro Ada development environment and toolset.

AdaCore can assist customers in developing ASIS-based tools to meet their specific needs, as well as develop such tools upon request.

Typical ASIS-for-GNAT applications include:

- Static analysis (property verification)
- Code instrumentation
- Design and document generation tools

- Metric testing or timing Tools
- Dependency tree analysis tools
- Type dictionary generators
- Coding standard enforcement tools
- Language translators (e.g., to CORBA IDL)
- Quality assessment tools
- Source browsers and formatters
- Syntax directed editors

GNAT2XML provides the same information as ASIS but allows users to manipulate it through an XML tree.

3.6. Static Verification - CodePeer

CodePeer is an Ada source code analyzer that detects run-time and logic errors. CodePeer assesses potential bugs before program execution, serving as an automated peer reviewer, helping to find errors efficiently and early in the development life-cycle. It can also be used to perform impact analysis when introducing changes to the existing code, as well as helping vulnerability analysis. Using control-flow, data-flow, and other advanced static analysis techniques, CodePeer detects errors that would otherwise only be found through labor-intensive debugging.

CodePeer can analyze programs written in full Ada (including all the features of Ada 2012) and does not rely on a particular language subset having been used. It is therefore suitable for analysis and assurance of existing code bases, and maintaining discipline for new and modified code.

As a stand-alone tool, CodePeer can also be used with projects that do not use GNAT Pro for compilation.

Early Error Detection

CodePeer's advanced static error detection finds bugs in code by mathematically analyzing every line of code, considering every possible input and every path through the program. CodePeer can be used very early in the development life cycle to identify problems when defects are much less costly to repair. It can also be used retrospectively on existing code bases, to detect latent vulnerabilities.

CodePeer can be used from within the GNAT Pro development environment, or as part of a continuous integration regime. It can detect several of the "Top 25 Most Dangerous Software Errors" in the Common Weakness Enumeration: CWE-120 (Classic Buffer Overflow), CWE-131 (Incorrect Calculation of Buffer Size), and CWE-190 (Integer Overflow or Wraparound). See [14] for more details.

CodePeer has been certified by the MITRE Corporation as a "CWE-Compatible" product [15].

3.7. Static Verification - SPARK Pro

SPARK Pro offers the ultimate toolset for high-integrity development. Through the discipline of the language subset, the SPARK Pro tools are able to offer verification that combines speed, flexibility, depth and soundness. Adoption of the language subset means that SPARK Pro is best suited for new high-assurance code (including situations where the existing code is at a lower assurance level and is written in full Ada or other languages such as C) or projects where the existing high-assurance coding standard is sufficiently close to SPARK to ease transition.

Powerful Static Verification

The SPARK language supports a wide range of static verification techniques. At one end of the spectrum is basic data- and control-flow analysis, i.e., exhaustive detection of errors such as attempted reads of uninitialized variables, and ineffective assignments (where a variable is assigned a value that is never read). For more critical applications, dependency contracts can constrain the information flow allowed in an application. Violations of these contracts – potentially representing violations of safety or security policies – can then be detected even before the code is compiled.

In addition, SPARK supports mathematical proof and can thus provide high confidence that the software meets a range of assurance requirements: from the absence of run-time exceptions, to the enforcement of safety or security properties, to compliance with a formal specification of the program's required behavior.

Minimal Run-Time Footprint

For the most secure systems (for example, embedded cryptographic devices), a developer has to worry about and justify the presence of *all* the code in a delivered system. Guidance talks of "minimizing the trusted computing base", which really means just making the delivered system as small as possible. There is also the problem of Commercial Off-the-Shelf (COTS) components: if a system uses a COTS library or operating system, then how are these to be evaluated or verified without the close (and probably expensive) cooperation of the COTS vendor?

For the most critical embedded systems, SPARK supports the so-called "Bare-Metal" development style, where SPARK code is running directly on a CPU with little or no COTS libraries or operating system at all. SPARK is also designed to be compatible with GNAT Pro's Zero FootPrint (ZFP) run-time library. In a Bare-Metal/ZFP development, every byte of object code can be traced to the application's source code, and accounted for. This can be particularly useful for systems that must withstand evaluation by a national technical authority or regulator.

SPARK code can also run on top of a full Ada run-time library and a commercial desktop operating system or anything in-between, but the choice is left to the system designer, not imposed by the language.

3.8. Dynamic Analysis Tools

GNATtest

The GNATtest tool helps create and maintain a complete unit testing infrastructure for complex projects. Based on AUnit, it captures the simple idea that each visible subprogram should have at least one corresponding unit test. GNATtest takes a project file as input, and produces two outputs:

- The complete harnessing code for executing all the unit tests under consideration. This code is generated completely automatically.

- A set of separate test stubs for each subprogram to be tested. These test stubs are to be completed by the user.

GNATtest handles Ada's Object-Oriented Programming features and can help verify tagged type substitutability (the Liskov Substitution Principle), which can be used to demonstrate consistency of class hierarchies.

GNATemulator

GNATemulator is an efficient and flexible tool that provides integrated, lightweight target emulation.

Based on the QEMU technology, a generic and open-source machine emulator and virtualizer, GNATemulator allows software developers to compile code directly for their target architecture and run it on their host platform, through an approach that translates from the target object code to native instructions on the host. This avoids the inconvenience and cost of managing an actual board, while offering an efficient testing environment compatible with the final hardware.

There are two basic types of emulators. The first can serve as a surrogate for the final hardware during development for a wide range of verification activities, particularly those that require time accuracy. However, they tend to be extremely costly, and are often very slow. The second, which includes GNATemulator, does not attempt to be a complete time-accurate target board simulator, and thus it cannot be used for all aspects of testing. But it does provide a very efficient and cost-effective way to execute the target code very early in the development and verification processes. GNATemulator thus offers a practical compromise between a native environment that lacks target emulation capability, and a cross configuration where the final target hardware might not be available soon enough or in sufficient quantity.

GNATcoverage

GNATcoverage is a dynamic analysis tool that analyzes and reports program coverage. GNATcoverage can perform coverage analysis at both the object code level (instruction and branch coverage), and the source code level for Ada or C (Statement, Decision, and Modified Condition/Decision Coverage - MC/DC).

Unlike most other technologies, GNATcoverage is nonintrusive: it works without requiring instrumentation of the application code. Instead, the code runs directly on an instrumented execution platform, such as GNATemulator, Valgrind on Linux, or on a real board monitored by a probe.

See [16] for more details on the underlying technology.

3.9. Integrated Development Environments (IDEs)

GNAT Programming Studio (GPS)

GPS is a powerful and simple-to-use IDE that streamlines software development from the initial coding stage through testing, debugging, system integration, and maintenance. GPS is designed to allow programmers to exploit the full capabilities of the GNAT Pro technology.

Tools

GPS's extensive navigation and analysis tools can generate a variety of useful information including call graphs, source dependencies, project organization, and complexity metrics, giving the developer a thorough understanding of a program at multiple levels. It allows interfacing with third-party Version Control Systems, easing both development and maintenance.

Robust, Flexible and Extensible

Especially suited for large, complex systems, GPS can import existing projects from other Ada implementations while adhering to their file naming conventions and retaining the existing directory organization. Through the multi-language capabilities of GPS, components written in C and C++ can also be handled. GPS is highly extensible; additional tools can be plugged in through a simple scripting approach. It is also tailorable, allowing various aspects of the program's appearance to be customized in the editor.

Easy to Learn, Easy to Use

GPS is intuitive to new users thanks to its menu-driven interface with extensive online help (including documentation of all the menu selections) and “tool tips”. The Project Wizard makes it simple to get started, supplying default values for almost all of the project properties. For experienced users, GPS offers the necessary level of control for advanced purposes; e.g., the ability to run command scripts. Anything that can be done on the command line is achievable through the menu interface.

Remote Programming

Integrated into GPS, Remote Programming provides a secure and efficient way for programmers to access any number of remote servers on a wide variety of platforms while taking advantage of the power and familiarity of their local PC workstations.

Eclipse support - GNATbench

GNATbench is an Ada development plug-in for Eclipse and Wind River’s Workbench environment. The Workbench integration supports Ada development on a variety of VxWorks real-time operating systems. The Eclipse version is primarily for native applications, with some support for cross development. In both cases the Ada tools are tightly integrated.

GNATdashboard

GNATdashboard serves as a one-stop control panel for monitoring and improving the quality of Ada software. It integrates and aggregates the results of AdaCore’s various static and dynamic analysis tools (GNATmetric, GNATcheck, GNATcoverage, CodePeer, SPARK Pro, among others) within a common interface, helping quality assurance managers and project leaders understand or reduce their software’s technical debt, and eliminating the need for manual input.

GNATdashboard fits naturally into a continuous integration environment, providing users with metrics on code complexity, code coverage, conformance to coding standards, and more.

4. Security Vulnerabilities and Their Mitigation

This chapter considers a number of specific and high-profile software vulnerabilities, inspired by the CWE/SANS "Top 25 Most Dangerous Software Errors" [17], and discusses how each can be prevented or mitigated using Ada, SPARK, and AdaCore's tools.

Some vulnerabilities are *universal* in that *all* software should be free of all occurrences—buffer overflow would be a good example, since all programs should be free of all buffer overflows, regardless of the particular application's requirements or operational domain.

Many vulnerabilities are in some way *application specific* in that they may or may not be a problem, depending on the application's particular security requirements and operational environment.

Related CWE identifiers are given in each sub-section. A more detailed list of other CWEs that are handled by Ada and/or AdaCore tools is presented in Appendix A.

4.1. Data Validation

Related CWEs

CWE	Short description	Notes
20	Improper Input Validation	Plus all children and variants
1019	Validate Inputs	Plus all children and variants

Vulnerability

Missing or incorrect validation of input data remains one of the most common security vulnerabilities in software. This is an application-specific vulnerability, since exactly what does or doesn't constitute "valid" input data is highly dependent on an application and its security requirements.

Ada offers a range of protections from these problems, from basic dynamic checks at run time to advanced static analysis and proof techniques.

Dynamic Mitigation

At the most basic level, Ada has always offered run-time range checking for scalar values. If a check fails at run time, then an exception is raised rather than allowing the execution of the program to become undefined. This offers protection against common defects such as integer range violations, buffer overflows, arithmetic overflow and division by zero. For example, any attempt to store an integer value outside the range (-180 .. 180) for an angle, or a real value outside the range (0.0 .. 10000.0) for a length in the following example will raise an exception at run time. Similarly, a `Data` value whose `Kind` is `Angle_Data` cannot be mistakenly interpreted as a value whose `Kind` is `Length_Data` (i.e., an `Angle` bit pattern cannot be interpreted as a `Length`) when using the discriminated unions of Ada; such an error would raise an exception.

```
type Angle is new Integer range -180 .. 180;
type Length is new Float range 0.0 .. 10_000.0;
type Datatype is (Angle_Data, Length_Data);
type Data (Kind : Datatype) is record
   case Kind is
      when Angle_Data =>
         A : Angle;
      when Length_Data =>
         L : Length;
   end case;
end record;
```

Ada 95 added a special attribute X'Valid for any scalar object X. This returns True if and only if the raw bit-pattern present in memory is a valid value for the type of the object *and* satisfies any subtype constraint or predicate (if present). This is more powerful than a simple “range check”, because it applies to types with complex representations such as floating-point or enumeration types with non-contiguous values. Further, the evaluation of X'Valid can never itself become undefined or raise an exception, so it provides a way to “peek” at incoming data to see if it’s

OK before proceeding. It also works with the ZFP run time and SPARK, where exceptions are excluded anyway.

GNAT Pro adds an attribute X'Valid_Scalars that can be additionally applied to composite types like records and arrays. This applies the correct X'Valid test recursively to *all* the components of a composite object, and only returns True if they are all OK and also satisfy any subtype constraint/predicate. For example, an input value of type Data from the previous example could be validated by evaluating X'Valid_Scalars, which will check that X.Kind is a valid Datatype value, and depending on this value, that either X.A is a valid angle or X.L is a valid length.

In terms of run-time verification, GNAT Pro also offers an extended validity checking mode. This instructs the compiler to make worst-case assumptions about data validity and assume, for example, that memory might have been corrupted at any point, so it *automatically* inserts a validity test for all objects, every time they are read. This comes with a noticeable performance penalty, but offers the most protection. As noted in section 3.3, this mode can be particularly useful in combination with automated *Fuzzing* (essentially random input testing), since the extended validity checks spot a problem sooner rather than later. GNAT Pro also supports special pragmas that instruct the compiler to initialize scalar objects to a value which is known to be *invalid* and will therefore always fail a validity test on first access. This offers an easy way to spot uninitialized values at run time, protecting against another pernicious undefined behavior.

Static Mitigation

The GNAT Pro compiler can detect some violations of data constraints that do not depend on the flow of control and analysis of calls. In such cases, it issues a warning that an exception will be raised at run time if that code is executed. Similarly, it can detect some simple cases of reading an uninitialized variable.

CodePeer can go further by analyzing values and relations between variables in a fully flow-sensitive and interprocedural analysis. CodePeer offers a range of analyses that protect from data validity problems and implements a form of *data-flow analysis* that statically detects uninitialized variables.

SPARK Pro goes further in a number of ways. Firstly, SPARK Pro offers a completely static verification for the absence of all undefined behavior, run-time errors and exceptions. In SPARK it's possible to prove that none of Ada's predefined run-time checks will ever fail for any program executions.

In the most general sense, subprograms in Ada and SPARK can also include *precondition* contracts that can specify arbitrary validity requirements on their parameters, which can be as permissive or as strict as is required by the designer. These can be checked at run time, or by static analysis, or both.

4.2. Native Code Injection

Related CWEs

CWE	Short description	Notes
94	Code Injection	
95	Eval Injection	
96	Static Code Injection	
97	Improper Neutralization of Server-Side Includes	
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Plus all children and variants
470	Unsafe Reflection	

Vulnerability

This section specifically deals with the problem of malicious injection of Ada or machine code. This is a universal vulnerability. Injection of code in other languages (e.g., scripting languages or SQL) can be application specific, so is considered elsewhere. Two cases are covered here—injection of Ada code itself, and injection of compiled machine code.

Mitigation—Ada code injection

Ada has always been a compiled language. There is no "reflection" or "eval"-like construct so it is impossible for Ada source code to be maliciously inserted and/or interpreted at run time.

Furthermore, Ada programs (especially for embedded systems) can be statically linked, and therefore are not susceptible to "DLL spoofing" or other attacks relating to shared libraries or dynamic linking. Finally, compiled Ada code is always executable from a read-only memory (such as ROM or a FLASH device) so can be further protected from tampering.

Mitigation—Machine Code Injection

This remains a common attack against unsafe programming languages and defective code. In short, a buffer over-write defect results in overwriting the stack memory with malicious data, which is actually the attacker's machine code. The return address on the stack is also manipulated to force control to jump to the malicious code. There are variants on this theme (particularly the so-called "return oriented programming" (ROP) family of attacks) but they all rely on a buffer overflow defect as the initial point of entry.

Ada is strongly protected from this class of vulnerability, owing to run-time checking of all array accesses, exception handling, and the strong forms of static analysis offered by both CodePeer and SPARK Pro. See section 4.1 for more details, since the same dynamic and static verification techniques that apply to data validity also apply to buffer overflows and other defects that lead to code injection.

4.3. Denial of Service

Related CWEs

CWE	Short description	Notes
400	Resource Exhaustion	Plus all children and variants
606	Unchecked Input for Loop Condition	
674	Uncontrolled Recursion	

Vulnerability

"Denial-of-Service" has become a broad term that refers to any form of attack that prevents a computer system from fulfilling its intended role and service. This is an application-specific vulnerability, since some systems can "fail secure" while others might have onerous requirements for continuity of service and availability.

Three sub-classes of attack are worthy of mention:

- *Forced immediate termination*. An attacker crafts input data that is designed to make a target system immediately terminate or "crash". A good example would an input that provokes an unhandled exceptional situation such as a division-by-zero.
- *Termination through resource starvation*. In this case, the attacker still causes the target system to terminate unexpectedly, but does so by deliberately exhausting its resources. For example, a flood of requests from the attacker causes the target system to "run out of memory" and eventually terminate.
- *Starvation*. An attacker floods a system with legitimate-looking, but bogus requests. The system continues to "work" until this enormous load, but legitimate users are "starved out".

Dynamic Mitigation

For some systems, termination in a known "secure state" might be acceptable. This kind of "fail secure" behavior is supported by Ada through run-time exception handling. A top-level "catch all" handler can be inserted into the main program and each task type or object; the handler can bring the system to a safe and/or secure state before allowing the system to terminate.

A second option is similar, but the system can implement some sort of "graceful degradation" and switch to a simpler mode of operation. Other options include reverting to a backup system, or executing a hardware-based "reset" to bring the system to a known state.

Static Mitigation

Where continuity of service is important, for example in communications and real-time control applications, both CodePeer and SPARK Pro offer strong protection. If programs can be statically shown to be free from all run-time errors, then they are effectively *crash proof* in the face of arbitrary input data.

To protect against possible non-termination of loops, CodePeer issues warnings when it detects that a loop may not terminate. SPARK Pro goes beyond mere warnings and can prove that loops terminate if the user specifies a loop *variant*—an expression that can be shown to increase or decrease towards a constant bound for every iteration of the loop.

SPARK is also immune from starvation or exhaustion of heap-based memory, since SPARK can be compiled without use of a heap. The GNATstack tool can also be used to show that SPARK programs will never run out of stack memory.

In all these scenarios, a "defense in depth" approach is appropriate where, for example, SPARK Pro might be used to statically eliminate run-time errors, but the system is still compiled with run-time checks, extended validity checking, and a top-level "catch all" exception handler.

4.4. Information Leak

Related CWEs

CWE	Short description	Notes
120	Classic Buffer Overflow	
121	Stack-based Buffer Overflow	
122	Heap-based Buffer Overflow	
125	Out-of-bounds Read	
126	Buffer Over-read	
127	Buffer Under-read	
200	Information Exposure	Plus all children and variants
514	Covert Channel	Plus all children and variants
665	Improper Initialization	Plus all children and variants

Vulnerability

These vulnerabilities form a general class of problems where information is seen to go where it shouldn't. Three sub-classes of this problem arise:

Mitigation—Programming Defects

This is a universal vulnerability. Simple programming defects can cause information to flow in unexpected ways. For example:

- An uninitialized variable can result in a read from memory of a value that has been "left over" on the stack from a previous computation.
- An unchecked buffer over-read can yield sensitive or incorrect information. The notable "Heartbleed" vulnerability in the OpenSSL library was of this class.

As noted earlier, Ada offers strong protection from these classes of vulnerabilities. Uninitialized variables can be tackled with GNAT Pro's enhanced validity checking modes, and buffer overflows are always prevented by run-time checks in Ada. CodePeer and SPARK offer static protection against both of these defect classes.

Mitigation—Algorithmic Defects

If a programmer simply implements "the wrong code", then unintended information flow can result. For example, if a function is supposed to be computed from two input variables X and Y, but the programmer mistakenly computes the result from X, Y and Z, then an observer might be able to deduce something about the value of Z, which might be a security vulnerability if Z is some sensitive value like a cryptographic key.

The exact nature of these vulnerabilities is highly application specific, but Ada and SPARK offer some protection through the use of contracts. A *Depends contract*, for example, could specify that the function result *must be* computed from X and Y *and not Z* and this specification is verified by the SPARK Pro tools with *information-flow analysis*.

Contracts can also be used to specify, for example, that data of different classification (e.g. "Unclassified" and "Top-Secret") shall not be mixed in a single computation. Again, SPARK Pro offers strong verification for such properties.

Mitigation—Side Channels

So-called "Side" or "Covert" channels exploit devious and unusual observations of a program's execution to deduce its internal state. These include:

- Timing-based attacks. Observation of a program's *execution time* can yield information on its internal state and variables.
- Power. Observation of the electrical power consumption of a computer can divulge what's going on internally.
- Electro-magnetic emissions.
- Acoustic (sound) emissions.
- Other things that no-one has even thought of yet...

These attacks are extremely difficult to prevent using software techniques alone. SPARK offers some assistance, since it is designed to be amenable to timing analysis, and its information flow analysis engine could be used to detect where execution time depends on a particular critical variable.

4.5. Improper use of API

"Entia non sunt multiplicanda praeter necessitatem"
["Entities are not to be multiplied beyond necessity"]
– William of Ockham, 14th Century.

or, put another way

"When in doubt, leave it out..."
– Joshua Bloch, Google.

Related CWEs

CWE	Short description	Notes
440	Expected Behaviour Violation	
559	Often Misused: Arguments and Parameters	
628	Function Call with Incorrectly Specified Arguments	
648	Incorrect Use of Privileged APIs	
749	Exposed Dangerous Method or Function	

Vulnerability

The design of reusable, general, and *error-tolerant* APIs remains one of the core skills of a software designer. (See Joshua Bloch's lecture [18] for an overview of this topic.)

For secure systems, the unintentional misuse of cryptographic or communications APIs and libraries is a regular source of defects and headlines.

A core issue is the expressive power, precision and abstraction with which a programming language allows an API specification to be defined. Many APIs need to express usage rules—what a user should and shouldn't do to use the API properly—and if these rules are specified formally then they can be checked either at run time or using static analysis.

Mitigations

Once again, Ada's contracts offer strong support. Preconditions express exactly the conditions under which a particular operation may be invoked, and can even be used to express ordering constraints on operations (e.g. "operation X must be invoked before either of operations Y or Z.") Similarly, postcondition contracts can express exactly what operations promise to do (and not do.)

Increasingly, such contracts are being added to parts of the standard Ada library to increase the strength of their specifications, and as a result their usage can be verified statically by the CodePeer and SPARK Pro tools. This is the case for example for the standard numerical library shipped with GNAT Pro. Some freely available libraries have also started to appear in SPARK, such as standard cryptographic algorithms [19].

4.6. Weak or No Crypto

Related CWEs

CWE	Short description	Notes
326	Inadequate Encryption Strength	Plus all children and variants
327	Use of a Broken or Risky Cryptographic Algorithm	Plus all children and variants
338	Use of a Cryptographically Weak PRNG	

Vulnerability

These are clearly application specific. The appropriate use of cryptographic algorithms depends on an application's precise needs for confidentiality, authentication and integrity of data, plus the perceived capability and threat owing to attackers.

The "strength" of cryptographic algorithms can depend on the algorithm chosen, the key length employed, and the quality of random numbers that are used in the generation of critical values such as keys and "nonce" values.

Mitigations

Ada can help to some extent through its system of strong types. A good design approach would be to declare distinct and *incompatible* types for unencrypted and encrypted data, so that they cannot be confused or used in the wrong context. For example:

```
type Plaintext_Buffer is private;

type Encrypted_Buffer is private
```

Even though, under the hood, these types might both represent an unstructured sequence of bytes, they are distinct and incompatible from the point of view of the client. For example, a procedure that outputs an encrypted buffer to a communications channel might be declared:

```ada
procedure Send_Buffer (B : in Encrypted_Buffer);
```

This procedure *cannot* send a Plaintext_Buffer and any attempt to do so would be rejected by the compiler.

If it is necessary to encrypt a plaintext buffer to produce an encrypted buffer, then a *single* function can do this *and* use contracts to enforce the strength of the key. For example:

```ada
type Key is limited private; -- no copying permitted!

function Strength_Of (K : in Key) return Natural;

function Encrypt (Data : in Plaintext_Buffer;
                  K    : in Key) return Encrypted_Buffer
  with Pre => Strength_Of (K) >= 256;
```

Note that the Key type is declared *limited private* so that objects of that type cannot be copied by assignment—another built-in feature of Ada that is ideal for such sensitive types.

4.7. Failure to erase sensitive data

Related CWEs

CWE	Short description	Notes
14	Compiler Removal of Code to Clear Buffers	
226	Sensitive Information Uncleared Before Release	Plus all children and variants
733	Compiler Optimization Removal or Modification of Security-critical Code	

Vulnerability

This vulnerability concerns the need to *erase* (or "sanitize") sensitive data, such as cryptographic keys, after they have been used, to prevent unintentional leak or exposure of that some time later.

This is a complex issue that spans application-specific needs across to highly technical implementation details. At the high level, applications need to define exactly what data is "sensitive" in the first place, and how much protection is required. At the low end of the spectrum, just "writing zeros" into memory might suffice. At the high end, it might be necessary to physical destroy hard disks and memory chips.

As a software programming challenge, this problem is more complex than it looks. Issues include data validity ("all zeroes" might be illegal or invalid), how to stop a compiler optimizing away the sanitizing code, avoiding explicit or implicit copying of sensitive data, coping with the complexities of data caches and memory devices, and so on.

Mitigations

Ada provides some specific support in this area. Its *limited* types are ideal for sensitive data since they cannot be copied by assignment, and are always passed by reference at run time. Ada 95 also introduced a special pragma `Inspection_Point` which serves to forbid "dead store elimination" in the compiler for a particular object at a particular place, thus ensuring that a final "sanitizing assignment" is not removed.

SPARK Pro offers some static verification support through information-flow analysis, since a final sanitizing assignment is reported as an expected anomaly (because it does not contribute to the functional behavior of the program), and can be justified as such.

A full description of this problem and how it was solved in a particular project can be found in [20].

4.8. Authentication and Authorization

Related CWEs

CWE	Short description	Notes
284	Improper Access Control	Plus all children and variants

Vulnerability

This large family of vulnerabilities is highly application specific. What is or is not “authorized” for a user of a system depends on the system, its environment, and the threat model that is expected.

Mitigations

In the broadest terms, systems should be designed with the *least privilege* principle in mind, restricting the most important or risky operations to the fewest users and operational scenarios.

Given a strong design along those lines, these concepts can be encoded in Ada using types and contracts. A simple model might map a user’s ID onto some ordered value of authorization:

```
type User_ID is limited private;

type Authorization is (None, Low, High);

function Authorizaton_Of (U : User_ID) return Authorization;
```

Sensitive operations can then use a precondition to restrict access, such as:

```
procedure Sensitive_Operation (Data : in out Encrypted_Buffer;
                               User : in     User_ID)
  with Pre => Authorization_Of (User) = High;
```

Operations that “escalate” a particular user’s authorization (from “Low” to “High” for example) could be strictly controlled and verified using Ada’s types and the appropriate combination of static and dynamic checking.

5. Industrial Scenario Examples

5.1. Overview

This chapter presents a number of security-related scenarios that may arise in real-world projects. Each opens with a description of the context and the security issue, and then shows how either Ada or SPARK, in conjunction with the relevant AdaCore tool(s), can contribute. Each scenario is illustrated with one or more examples, drawn from experience with customers and industrial projects.

5.2. Scenario 1: Identifying and repairing security vulnerabilities in existing Ada codebases

In this scenario an existing system written in Ada has to be analyzed for security vulnerabilities, perhaps because of regulatory oversight or commercial/corporate obligations. This is typical of closed systems which become exposed to new threats by connecting them to other systems (ad-hoc networks, Internet, etc.)

The recommended approach comprises two steps:

1. Use GNAT Pro and inspect the compiler warnings.

   Even when GNAT Pro is not the main compiler on a project, it can still be used as a basic static analysis tool. GNAT Pro flags around 50 different classes of warnings which represent over 130 warning messages. Experience shows that a careful selection of warnings combined with a fix-all-warnings policy can significantly increase the quality of a code base. Warnings that cannot be fixed can be justified with pragma Warnings Off. Warnings can be treated as errors (thus stopping compilation) by enforcing the fix-all-warnings policy with switch -gnatwe.

2. Submit the code base to CodePeer for analysis.

   CodePeer can exhaustively detect all occurrences of many vulnerabilities from the CWE list: CWE-120 (“Classic Buffer Overflow”), CWE-190 (“Integer Overflow or Wraparound”), CWE-476 (“NULL Pointer Dereference”), CWE-571 (“Expression is Always True”), etc. Users have the choice to opt for an exhaustive report of all potential vulnerabilities (using the -level max switch) or, more commonly, to adjust the level of analysis to their needs, balancing soundness with the effort required to review all messages.

   CodePeer can also include GNAT warnings in its messages by using the switch -gnat-warnings. This is particularly relevant for projects that do not use GNAT as the compiler, or else use an older version of GNAT that may lack some of the newer warnings. The GNAT warnings selected are described in the CodePeer User’s Guide.

Example of Scenario 1 – GNAT Pro Compiler

GNAT Pro is the compiler for Ada developed by AdaCore, based on the GCC compiler architecture. The front end of GNAT Pro is written in Ada. It is a large software component with 458 units, 370 ksloc, and has been in development since 1992. The front end is compiled with warnings-as-errors (-gnawe) and a large set of warnings enabled (-gnatwa).

Since 2017, AdaCore has been running CodePeer on the GNAT Pro front end, with a fix-all-messages policy. These runs have resulted in the detection of a number of errors in the code, as well as code quality issues (e.g. dead code) which are opportunities for refactoring. CodePeer is run at level 1 for fast execution (less than 10 minutes on a developer machine) while minimizing false alarms. Remaining false alarms are justified with pragma Annotate in the code. CodePeer runs have been integrated in the continuous building environment and nightly regression testing.

5.3. Scenario 2: Ada software development practices for increasing security

This case covers a majority of ongoing software developments in Ada, where the strengths of the Ada language are combined with the AdaCore toolset to deliver high-quality software with lower error density than with other languages/toolsets.

Of particular importance, especially for high-integrity development, are the Ada features for encapsulation (packages / private types), reuse (generics), control of representation (data size and layout, addresses), strong typing (type constraints, predicates, invariants), and contract-based programming (preconditions and postconditions). Specific guidance is available for the use of Object-Oriented Programming in Ada [12].

Tools that are relevant in most contexts for high-integrity development are the GNAT Pro compiler for warnings and style checking, the GNATcheck coding standard checker, GNATmetric for metrics computation, GNATstack for memory usage analysis, GNATtest for test harness generation, and CodePeer for static analysis.

Example of Scenario 2 – Ada Web Server

The Ada Web Server (AWS) is an Ada implementation of the HTTP/1.1 protocol. It is a library that can be embedded in an application to allow communication with modern web browsers. AWS supports HTTPS (secure HTTP) using SSL. This is based on either OpenSSL or GNUTLS, two open-source SSL implementations.

Because AWS is security sensitive, special care is taken in its code to state explicitly the constraints that should be respected for the program to operate without errors, using Ada contracts on types and subprograms. For example, AWS code deals with time-zone string representation in many places. The code uses a predicate on this type to enforce that this representation remains valid:

```
subtype Time_Zone_String is String with
 Dynamic_Predicate =>
  (Time_Zone_String'Length = 0
     or else
  (Time_Zone_String'Length = 5
     and then
   Time_Zone_String (Time_Zone_String'First) in '-' | '+'
     and then
   Time_Zone_String (Time_Zone_String'First + 1) in '0' .. '2'
     and then
   Time_Zone_String (Time_Zone_String'First + 2) in '0' .. '9'
     and then
   Time_Zone_String (Time_Zone_String'First + 3) in '0' .. '5'
     and then
   Time_Zone_String (Time_Zone_String'First + 4) in '0' .. '9'
  ));
```

In the same vein AWS uses a Hex_String type which contains only numbers and letters from 'a' to 'f'.

AWS also uses preconditions and postconditions on many subprograms. For example, when building an object containing a response to be sent back to the Web browser, the postcondition ensures at a minimum that the Build routine does not return an empty object, and that the Status_Code of the response is set according to the corresponding parameter:

```
function Build
  (Content_Type    : String;
   UString_Message : Unbounded_String;
   Status_Code     : Messages.Status_Code := …;
   Cache_Control   : Messages.Cache_Option := …;
   Encoding        : Messages.Content_Encoding := …)
   return Data
with Post => not Is_Empty (Build'Result)
   and then Response.Status_Code (Build'Result) = Status_Code;
```

As another example, in the Session API, AWS uses a postcondition to ensure that the value of a session is empty if the corresponding key is unknown:

```
function Get (SID : Id; Key : String) return String with
 Post => (not Exist (SID, Key) and then Get'Result'Length = 0)
         or else Exist (SID, Key);
```

The benefit in expressing these constraints as type predicates and subprogram contracts is that they can be checked at run time, instead of informal comments as would be used in other languages. Another benefit is that these explicit contracts replace defensive programming in a way that makes it clear to clients of the API what is expected.

5.4. Scenario 3: Secure Design through SPARK

This scenario illustrates a high-security system where maximum assurance is required. Such systems often contain few or no COTS components and can be both embedded and feature a "bare metal" implementation style with minimal or no operating system support.

At the high-end, a "bare metal" implementation, the use of SPARK and the Zero FootPrint (ZFP) Ada run-time library offer the ability to account for every byte of object code in the finished product.

For such systems, the recommended approach is to use SPARK in fully constructive mode with Verification-Driven Design (see section 2.4) to set objectives for each subsystem or module. All code should be proven free from run-time errors, and critical modules should be proven to satisfy application-specific security properties.

Example of Scenario 3 – the Muen Kernel

Muen [21] is a high-assurance hypervisor for the x86_64 architecture, with the most critical components designed and verified using SPARK. Muen is also a Separation Kernel and supports strict partitioning and security policy enforcement for its clients.

The Muen system, including all the code, is freely available under version 3 of the GNU Public License (GPL).

Examples of Scenario 3 – Crypto and Tokeneer

Several projects have used SPARK to develop critical cryptographic components, including cross-domain switches and reference implementations of cryptographic algorithms [19, 22, 23]. Some of these have been evaluated to the highest levels of assurance required by national regulators.

The Tokeneer project [24] was a demonstration of high-integrity development in SPARK, funded by the US National Security Agency (NSA). The Tokeneer code and all documentation have been released under a permissive license and are freely available for study and research.

5.5. Scenario 4: Support for Mixed Criticality Systems

This scenario covers systems with mixed assurance requirements, developed using a variety of technologies and programming languages.

The key is a sound security engineering and architectural design that separates critical from non-critical components, and that makes the most critical components as small as possible. The architecture should support a verification argument that top-level security requirements are indeed met.

Various implementation mechanisms can support such an architecture—distinct CPUs, multi-core CPUs, hypervisors and RTOSs can all offer the support required. The software might be several distinct "programs" which might be implemented in SPARK (for the most critical), Ada (for mission-critical components and infrastructure), C (for some low-level functions) and possibly other languages like C++, Java or Python for a user-interface component.

Example of Scenario 4

The MULTOS CA [8] formed the root certificate authority and key generation facility for the MULTOS smartcard operating system. The CA facility stored the *private* signing keys that were used to digitally sign certificates for MULTOS applications, and so were subject to an extraordinary level of physical, procedural, and computer-based security.

The software architecture was similar to that described above. Great care was taken to isolate security-critical functions in a single *security kernel* component that was constructed and verified using SPARK. In another architectural simplification, the system was designed so that there was no *concurrent* execution of security-critical functions. The system's software infrastructure was developed in Ada (using tasking for the non-critical, but naturally concurrent activities), while the GUI was implemented using C++ and the Microsoft Foundation Classes. A small number of C libraries were used, while a small amount of SQL code supported the system's internal database.

5.6. Scenario 5: Introducing Ada in a C project

The introduction of Ada and/or SPARK can be a challenge for some projects, especially those with a large code base in C or some other language. This section deals with C in particular, since it is commonly used for embedded systems development.

In fact, "mixed language" development with Ada, SPARK, C, and assembler is standard practice among AdaCore's customers. Projects often have libraries or components written in C which have long-standing provenance, so there is little desire (or technical merit) to rewrite them in Ada or SPARK [25].

Ada can be introduced into such an environment for new or modified subsytems, and linked with existing code. Ada has particularly strong support in this area, in terms of both language features and tooling. The Ada language definition devotes an entire annex to the matter of interfacing Ada code with other languages, with special sections for C and C++ among others.

The crucial step is to identify, isolate, and "wrap" C libraries, to provide an Ada interface for invoking them. Where a C library exports a function, for example, an Ada package would contain a corresponding subprogram declaration. Essentially, the idea is to produce a *specification* of the function in Ada, but to keep the *implementation* in the original language. The Ada specification can take advantage of Ada's strong types, parameter passing modes, and contracts to their full extent. This may involve careful study of the C library's documentation and implementation to understand fully what it does (and doesn't) promise to its clients. These properties and assumptions can be documented as contracts in the Ada code, but these are *formal* in that they can be used for both dynamic and static checking.

Once C libraries are "wrapped" in this fashion, they can be called from new Ada or SPARK code as expected. (It also works in the other direction: Ada and SPARK can be called from C.) The contracts on the library binding will be verified, either dynamically at run time, statically using CodePeer or SPARK Pro, or both.

AdaCore's tools support this style of development well. As a first-class member of the GCC family, GNAT Pro can compile Ada and C "out of the box", following all of the guidance in the interfacing annex of the Ada standard. Additionally, GNAT Pro supplies a "binding generator" that can automate the process of turning a C ".h" file into an equivalent Ada package specification. Many of AdaCore's other tools (e.g. GPS, GDB, and GNATcoverage) are also "multi-lingual" and work seamlessly with mixed-language code.

Example of Scenario 5: Industrial mixed-language system

This project is a large, critical system using a combination of Ada, SPARK, and C [26].

While the critical functions are implemented in SPARK, the user-interface is constructed using the X11/Motif framework and libraries which express their API in C. Therefore, a "binding layer" was constructed to connect the SPARK code, via C, to the underlying libraries. The C layer is subject to static analysis using the MISRA guidelines [27] and a MISRA checking tool.

Some effort was spent to document the *assumptions* that the SPARK and C code make about each other's behavior, and how these assumptions can be expressed as contracts and verified in practice. Further details of this project appear in [26], which is available from the authors.

6. Summing Up

Developing software that operates predictably and "does the right thing" in an overtly malicious environment is a high bar for software designers to overcome, but it can be done with a combination of engineering discipline, processes, languages and tools. While security remains a multi-faceted problem, the Ada and SPARK languages and AdaCore's tools provide some effective means to build software that truly matters.

From its earliest days, Ada has always emphasized the needs of high-integrity systems and, in particular, the verifiability of code. With the rising need for security in software, the strengths of Ada's design are gaining increased recognition and appreciation. Although other languages are trying to add support for high-integrity and secure programming, these are properties that need to be considered from the earliest stages of the language design, they cannot be grafted on afterwards. If and when those languages eventually arrive at that sweet spot, they will find Lady Ada already there waiting to greet them.

Ada's design also catalyzed the development of SPARK, which brings even greater emphasis on verifiability and *sound* static verification. The importance of soundness should not be underestimated—it allows some defect classes to be entirely prevented in the face of arbitrary input data. Static verification also saves money by reducing wasteful testing and rework later in the lifecycle, and stands a chance of freeing developers from an endless cycle of "test and patch."

In brief, Ada and SPARK and their associated tools stand out among current language technologies in addressing the two issues underling secure software:

- Ensuring that specific security functions are implemented correctly and enforce the required security policy, and
- Verifying that the rest of the software is free of vulnerabilities that could defeat the required security policy.

Developing secure software is by its nature a daunting problem, but Ada and SPARK can make it manageable.

A. CWE Mapping

Overview

This appendix focuses on the MITRE Corporation's Common Weakness Enumeration (CWE) and how the use of Ada and AdaCore technologies can address particular CWEs.

Several terms are used here with specific meanings. A language or tool is said to *prevent* a given CWE if:

- That CWE can be shown to be entirely absent from an application, and
- The argument is *sound* – i.e. there is confidence that *all* instances of that CWE have been prevented.

A language feature or tool is said to *mitigate* a CWE if either:

- The risk of that CWE occurring in an application is reduced, but perhaps not entirely eliminated, or
- Eliminating that CWE requires the user to remember to run a tool (e.g., CodePeer or SPARK Pro), and correctly interpret the results, or
- The user must formulate corrective action should a failure of that CWE be detected at run time—for example, the correct recovery action if a buffer overflow is detected via an exception, or
- Some combination of the above.

In the same way as the vulnerabilities covered in Chapter 4, some CWEs are *universal* in that *all* software should be free of all occurrences. Buffer overflow is an example, since all programs should be free of all buffer overflows, regardless of any individual application's requirements or operational domain.

Many CWEs are also *application specific*, depending on the application's particular security requirements and operational environment. For example, the class of CWEs commonly known as "SQL Injection" are highly relevant to web server applications that have a database supporting them, but are of no importance to a small embedded control system that has no “SQL Server” or database of any kind attached to it. Many of these CWEs can be prevented simply, but these require some thought on the part of the system designer.

Note that “CWE compliance” for an application is a highly domain- and application-specific concept. The list of CWEs also grows and evolves as weaknesses are discovered, so compliance with the CWE should not be seen as a one-off “box ticking” exercise. Rather, the CWEs should be considered as a *starting point* for developers, not an endpoint in itself.

Another important consideration is whether a CWE is mitigated by *static* or *dynamic* means. Ada offers substantial run-time checking for many CWEs via its built-in run-time checks and exception handling facilities. In contrast, tools like CodePeer and SPARK Pro offer *static* mitigation.

It is up to the designer to choose an appropriate mix of static and dynamic mitigation strategies. For some projects (for example, those working in a particularly harsh environment, such as a space-borne application) a combination of *both* static and dynamic mitigation might be appropriate.

CWEs prevented by Ada

The following table shows CWEs that relate to specific features of languages other than Ada—for example, a CWE that is particular to Java, and cannot affect an Ada program. Merely using Ada at all is sufficient to prevent these CWEs.

CWE Identifiers	Note
467, 484	Only affects C and C++
500	Only affects C++ and Java
520, 526	Only affects .NET languages
8, 9, 487, 555, 574,	Only affects Java
103, 104, 107, 108, 109, 110, 608	Only affects Struts framework

The next table shows a group of CWEs that reflect programming language problems and constructs that cannot affect Ada at all, but are not particular to any other specific language.

CWE Identifiers	Note
588	Unsafe pointer usage – not possible in Ada.
95	Unvalidated code in dynamic "eval" context – not possible in Ada.
481, 482	Confusion between assignment and comparison – not possible in Ada.
170	Improper null termination of Strings – not possible in Ada.
228, 229, 233, 237, 240 (and variants thereof)	Parameters missing/extra/confused – not possible in Ada owing to parameter passing rules and strong type checking.

CWEs Mitigated by Ada, CodePeer and SPARK

The following table lists CWEs by their identifier and short description, then shows how each is prevented or mitigated in columns with the following headings:

DM_Ada – Dynamically mitigated by Ada (using run-time exception handling for example.)

SM_CP – Statically mitigated by CodePeer.

SM_SP – Statically mitigated by SPARK Pro.

These tables only list "Base" CWEs, not "Class" or "Variant" CWEs.

CWE	Short description	DM_Ada	SM_CP	SM_SP
120	Buffer Overflow	Y	Y	Y
123	Write-what-where condition	Y	Y	Y
124	Buffer Under-write	Y	Y	Y
125	Out-of-bounds read	Y	Y	Y
128	Wrap-around error	Y	Y	Y
129	Improper validation of array index	Y	Y	Y
130	Improper handling of length parameter	Y	Y	Y
131	Incorrect calculation of buffer size	Y	Y	Y
136	Type errors	Y	Y	Y
137	Representation errors		Y	Y
188	Reliance on data layout			Y
190	Integer overflow or wrap-around	Y	Y	Y
191	Integer underflow or wrap-around	Y	Y	Y
193	Off-by-one error	Y	Y	Y
194	Unexpected sign extension	Y	Y	Y
197	Numeric truncation error	Y	Y	Y
252	Unchecked return value	Y	Y	Y
253	Incorrect check of function return value	Y	Y	Y
366	Race condition		Y	Y
369	Divide-by-zero	Y	Y	Y
456	Missing variable initialization		Y	Y
466	Return of pointer value outside expected range			Y
468	Incorrect pointer scaling			Y
469	Use of pointer subtraction to determine size			Y
476	Null pointer dereference	Y	Y	Y

562	Return of stack variable address	Y	Y	Y
682	Incorrect calculation	Y	Y	Y
786	Access before start of buffer	Y	Y	Y
787	Out-of-bounds write	Y	Y	Y
788	Access after end of buffer	Y	Y	Y
805	Buffer access with incorrect length	Y	Y	Y
820	Missing synchronization		Y	Y
821	Incorrect synchronization		Y	Y
822	Untrusted pointer access			Y
823	Out-of-range pointer offset			Y
824	Uninitialized pointer	Y	Y	Y
825	Expired pointer dereference			Y
835	Loop with unreachable exit		Y	Y

Ada Restrictions to CWE Mapping

As noted in section 3.1 Ada has a Restrictions pragma that allows particular language features to be forbidden from a program.

Some of these Restrictions remove entire classes of defect and vulnerability from programs at a stroke. The following table shows a mapping from Restrictions to CWE identifiers. Details on the exact meaning and effect of each Restriction are given in the GNAT Reference Manual.

Note to GNAT Pro users: If a project is not using the Restrictions pragma, then the list of Restrictions that could be applied can be generated using the GNAT Pro *Binder* tool's "-r" switch. In GPS, this switch can be enabled from the Project Properties dialog box, by selecting the Build / Switches / Binder menu entry and then checking the "List possible restrictions" checkbox.

Restriction Identifier	**CWEs prevented**
No_Allocators	122, 244, 415, 416, 467, 590, 761
No_Tasking or Max_Tasks => 0	362, 364, 366, 432, 479, 543, 558, 567, 572, 585, 662, 663, 820, 821, 828, 831, 833
No_Recursion	674
No_Exceptions	248, 396, 397, 460, 584, 600
No_Exception_Handlers	396, 584
No_Finalization	568, 583, 586
No_Streams	499
No_Unchecked_Conversion	197, 588, 704, 843
No_Wide_Characters	135, 176
No_Dependence	676 ("potentially dangerous functions" can be forbidden using this Restriction.)

B. Handling SQL Injection in Ada and SPARK

This appendix builds a worked example of how the common “SQL Injection” defect can be handled using Ada and SPARK.

SQL Injection is a particular instance of the more general vulnerability of “treating data as code”—something that developers should always be wary of, yet is extremely useful and common practice. In short, some “data” (coming from a user, another computer system, a network, etc.) arrives in a program and is composed or manipulated in some way to form “code” that is executed or interpreted. In this particular instance, the language in question is SQL which is executed by some database server.

These forms of “code injection” attacks have been at or near the top of the most-reported cyber security vulnerabilities for many years.

Some static analysis tools claim to find and report SQL Injection problems, using some form of “taint analysis”. This is where the flow of information from an input to an SQL query is tracked through a program, so that potentially suspicious (or “tainted”) queries can be reported. This approach is essentially heuristic—a tool is trying to assess if a particular user input *might* be tainted. This yields both false positives (spurious alarms) and false negatives (missing alarms).

The crux is that a general-purpose tool *cannot know* what does or doesn’t constitute a valid and secure SQL query without detailed knowledge of the target application’s security policy and requirements. This detail is inherently domain- and application specific, so there’s no way an “out of the box” tool can have such a built-in oracle.

This example shows how Ada’s *contracts* can be used to solve this problem.

In the following code, a very simple database API is expressed as an Ada package specification.

```
package DB
  with SPARK_Mode     => On,
       Abstract_State => State,
       Initializes    => State
is
   procedure Execute (SQL_Query : in      String;
                      Result    :     out Integer)
     with Global => (Input => State);
end DB;
```

For simplicity it is assumed that the Execute procedure takes a single String parameter (which is the SQL to be executed) and always returns an Integer value.

The contracts work as follows:

- SPARK_Mode means that the package is supposed to comply with the rules of the SPARK language subset.
- The Abstract_State contract allows the package to *name an abstraction* of some persistent state that is embodied in the package. In this case the name "State" represents the state of all the data in the database. It needs a name so that it can be referenced in later contracts.
- The Initializes contract specifies that the State data is to be considered to be well-defined and initialized when the program starts. In short, the database server is assumed to be "initialized" before the program begins execution.
- The Global contract on the Execute procedure signifies that, in addition to its "in" parameter, the procedure may *read* from the database State, but *not* write to it. This is useful, since it confirms that the Execute procedure *is not permitted* to change the state of the database.

So far, there is no explicit constraint on what constitutes a "valid" SQL Query. For example, a malicious client might call Execute with SQL_Query set to "DROP TABLE Customers;" which would clearly violate the intent.

A constraint is needed to ensure that the procedure Execute can only be called under specific circumstances when the query is valid. This is a *precondition* on the Execute procedure. Good software engineering practice entails *abstracting* that validity property as a *function*, and also adding a *Precondition* contract to procedure Execute, thus:

```
package DB
  with SPARK_Mode     => On,
       Abstract_State => State,
       Initializes    => State
is
   function Is_Valid (SQL_Query : in String) return Boolean;

   procedure Execute (SQL_Query : in     String;
                      Result    :    out Integer)
     with Global => (Input => State),
          Pre    => Is_Valid (SQL_Query);
end DB;
```

In Ada, that precondition will be evaluated every time that Execute is called, to ensure that all is well before the query is allowed to proceed.

If a main program tries to call Execute *without* checking Is_Valid first and chooses not to handle the resulting exception, then the result is predictable:

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE : failed precondition
from db.ads:11
```

If programmed correctly using Is_Valid as a defensive check, then the call will always be allowed to proceed. For example:

```
if DB.Is_Valid (Full_Query) then
   DB.Execute (Full_Query, Result);
else
   Result := 0;
end if;
```

No Free Lunch—Implementing Is_Valid

To build and execute a program like this, a designer would have to supply a legal body for the function Is_Valid. This poses something of a problem: what should the body be? How strict should it be? On the one hand Is_Valid could return True all the time—this would cause no violations but would be rather obviously unsound if a malicious query really did arrive. On the other hand, it could always return False—never allowing *any* SQL queries at all. This would be "secure" but offer a rather high false-positive rate, since all perfectly legal queries would be rejected.

Between these two extremes lies a "sweet spot" where Is_Valid implements *precisely* the security policy required by the system, yielding the "just right" balance of precision and soundness. However, this means that a security policy exists in the first place.

An important lesson for system designers is that technologies like Ada and SPARK can offer useful *mechanisms* in building secure systems, but they do not supply *policy*. That can only come from a knowledge of the system's application domain and requirements.

An example of Is_Valid

For simplicity this example assumes that the system's security policy exists and says that an SQL query is valid if:

- The lower-bound index of the query String is 1, and
- The String is at least seven characters long, and
- The first seven characters are "SELECT ", and
- The number of semi-colon characters (the statement terminator in SQL) is exactly 1, and
- The final character is a semi-colon.

For example, the query

"SELECT Name from Customers;"

would be deemed OK, while

"SELECT Name from Customers; DROP TABLE Customers;"

would be rejected.

This can be implemented in the body of package DB as follows:

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
package body DB is
   function Is_Valid (SQL_Query : in String) return Boolean is
   begin
      return    (SQL_Query'First = 1 and SQL_Query'Last >= 7)
       and then  SQL_Query (1 .. 7) = "SELECT "
       and then  SQL_Query (SQL_Query'Last) = ';'
       and then  Count (SQL_Query, ";") = 1;
   end Is_Valid;
end DB;
```

Going further with SPARK and Static Verification

This example has shown how Ada's contracts can offer precise facilities for *dynamic* verification of complex, domain-specific security properties like SQL Injection.

The SPARK language and tools go further, offering fully *static* verification of the same properties.

If a client package C1 calls DB.Execute twice, first without the correct defensive check, and later with the check correctly programmed, then the GNATprove tool might report:

```
$ gnatprove -Psql --report=all -u c1.adb
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
c1.adb:14:09: medium: precondition might fail
c1.adb:24:12: info: precondition proved
```

as expected.

Going further with Correct-by-Design SQL Queries

The above example shows how Ada contracts can be leveraged to provide a safety net against SQL injection by introducing a validity checking step for every SQL query, and how SPARK further provides static verification of this required check.

In practice however, a safe application would not build SQL queries using simple string operations in the first place. Instead, it would more likely use type-safe APIs like those provided by the library GNATCOLL.SQL, or higher level tools such as the GNATcoll Object Relational Mapper, to construct SQL queries whose syntax is fully constrained by Ada's static strong typing. The rendering of these queries into SQL, with safe handling of user-provided data (either through appropriate escaping, or even passing it out of band as separate parameters to the SQL execution back-end), can then be left to the API implementer.

References

[1] Ross Anderson. *Security Engineering*, 2nd Edition. Wiley, 2008. Also available at http://www.cl.cam.ac.uk/~rja14/book.html

[2] CERT CyberSecurity Engineering Site. https://www.cert.org/cybersecurity-engineering/

[3] Cyber Security Body of Knowledge website. http://www.cybok.org/

[4] European Cyber Security Organisation. *Overview of Existing Cybersecurity Standards and Certification Schemes v2*. Dec 2017. https://ecs-org.eu/documents/publications/5a31129ea8e97.pdf

[5] Daniel Jackson et al. *Software for Dependable Systems: Sufficient Evidence?* US National Academy of Sciences, 2007. https://www.nap.edu/catalog/11923/software-for-dependable-systems-sufficient-evidence

[6] Paul E. Black et al. *Dramatically Reducing Software Vulnerabilities*. NIST Report NISTIT 8151. January 2017. https://www.nist.gov/publications/dramatically-reducing-software-vulnerabilities

[7] National Cyber Security Centre. *UK National Cyber Security Strategy 2016-2021*. https://www.ncsc.gov.uk/document/national-cyber-security-strategy-ncss

[8] Roderick Chapman and Anthony Hall. *Correctness by construction: building a commercial secure system*. IEEE Software, vol. 19, no. 1, Jan/Feb 2002, pp. 18–25, DOI: 10.1109/52.976937. PDF available from the first author.

[9] ISO/IEC, *Ada Language Reference Manual*, 2012. Available at http://www.adaic.org/ada-resources/standards/ada12/

[10] John Barnes and Ben Brosgol, *Safe and Secure Software, an invitation to Ada 2012*, AdaCore, 2015.
Available at http://www.adacore.com/knowledge/technical-papers/safe-and-secure-software-an-invitation-to-ada-2012/

[11] John Barnes, *Programming in Ada 2012*, Cambridge University Press, 2014

[12] AdaCore, *High-Integrity Object-Oriented Programming in Ada*, 2013. Available at http://www.adacore.com/knowledge/technical-papers/high-integrity-oop-in-ada/

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

[14] Paul E. Black, Michael Kass, Michael Koo, Elizabeth Fong, *Source Code Security Analysis Tool Functional Specification*, NIST, 2011.

[15] CWE Compatible Product website.
http://cwe.mitre.org/compatible/compatible.html

[16] Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, Thomas Quinot, *Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework*, ERTS, 2010. PDF available here.

[17] CWE/SANS Top 25 Most Dangerous Software Errors.
http://cwe.mitre.org/top25/

[18] Joshua Bloch. *How to Design a Good API and Why it Matters.* Various on-line sources, including video here:
https://www.youtube.com/watch?v=aAb7hSCtvGw.

[19] A. Senier *et al. LibSPARKCrypto – A cryptographic library implemented in SPARK.* http://senier.net/libsparkcrypto/

[20] Roderick Chapman. *Sanitizing Sensitive Data: How to Get It Right (or at Least Less Wrong)*. Proc of Reliable Software Technologies – Ada Europe 2017. Vienna, Austria, June 2017. Springer LNCS Vol. 10300. DOI: 10.1007/978-3-319-60588-3_3. PDF available from the author.

[21] Muen – An x86/64 Separation Kernel for High-Assurance. https://muen.codelabs.ch/

[22] R. Chapman, E. Botcazou, and A. Wallenburg. *SPARKSkein: a formal and fast reference implementation of Skein*. Proc 14th Brazilian Symp on Formal Methods, Sao Paulo, Brazil, Sept 2011. Springer-Verlag LNCS, vol. 7021, pp. 16–27. DOI: 10.1007/978-3-642-25032-3_2. PDF available from the first author.

[23] Yannick Moy. *SPARKSkein – From Tour-de-Force to Run-of-the-Mill Formal Verification*. SPARK 2014 Blog Entry, 1st June 2015.

[24] Tokeneer project archive. http://www.adacore.com/tokeneer

[25] Johannes Kanig, Quentin Ochem, Cyrille Comar. *Bringing SPARK to C developers*. 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Jan 2016, Toulouse, France. https://hal.archives-ouvertes.fr/hal-01258395/

[26] J. Kanig, R. Chapman, C. Comar, J. Guitton, Y Moy, and E. Rhys. *Explicit Assumptions—A Prenup for Marrying Static and Dynamic Program Verification*. Proc Tests and Proofs 2014. Springer-Verlag LNCS, vol. 8570, pp. 142 – 157. DOI: 10.1007/978-3-319-09099-3_11

[27] MISRA. *Guidelines for the Use of the C Language in Critical Systems*, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013. http://www.misra.org.uk/

Index