Building secure software: Your language matters!

By Robert B.K. Dewar, PhD, and Roderick Chapman, PhD

Producing secure systems requires advance planning: You have to design security in from the start, rather than reactively patch software in response to vulnerability reports. This "security up front" approach demands programming languages that help prevent bugs and insecurities from being introduced into software in the first place, static analysis tools that catch errors early, and development techniques that can provide confidence in the correctness of the resulting system. The Ada and SPARK languages satisfy these requirements more easily than alternatives such as C, C++, and Java, and SPARK has advanced the state of the practice with its rigorous proof-based approach to system security verification.

Reliability is obviously important for any piece of software, but there are two domains where reliability is not just a desirable goal but an essential requirement. The first is safety-critical systems, where a defect can directly lead to loss of life or have other catastrophic consequences. Familiar examples are commercial aviation, medical instrumentation, and automobile control. The second is the area of security-critical systems. Here we demand not only completely reliable behavior in normal operation, but also resistance to deliberate attempts at subversion. High-profile areas include banking, power station control, and electronic voting.

Developing a secure system involves an end-to-end analysis of all hardware and software components, and an analysis of possible vulnerabilities caused by human links in the chain. We focus herein on the choice of programming language and related tools. Although this is just one of the issues that must be considered, its effect is pervasive. We will look in particular at Ada[1] and at the Ada-based SPARK language[2], which have been successfully used for safety-critical systems and are attracting growing interest in the security community.

Maintainability

Recognizing that in large, critical projects the major costs come from verification and maintenance, Ada and SPARK have been designed with an emphasis on ease of reading versus ease of writing. For example, a compact notation like the "++" operator in C and C++ may be convenient for writers, but its succinctness comes at the price of understandability, especially in contexts such as x[i++]. Since its effect can be obtained by alternative and clearer constructs with equivalent performance, "++" is not provided by Ada or SPARK. As another example, Ada and SPARK allow underscores in numeric literals as a way of making the value easily understood. The hexadecimal literal 16#FFFF_FFFF_FFFF_FFFF# in Ada and SPARK is readily seen to mean 2^{64}-1; the corresponding literal in C, C++, and Java would be 0xFFFFFFFFFFFFFFFF, which requires careful scrutiny by the reader to check that the number of digits is correct.

Modularity

Arguably the most important feature of Ada, and inherited by SPARK, is its modularization facility. The package construct, by partitioning a module clearly into a specification and a separately compiled implementation, helps developers to decompose complex programs into manageable pieces. A key aspect of developing a secure system is to build it from separate components where the correctness of each component can be individually verified, and where the interactions among the components are controlled. The separation of specification and implementation (see Listing 1) is important in achieving this goal, and is a point of contrast between a language like Ada or SPARK and other languages. In C++, the use of header files can be used to partially obtain this effect, but the language does not enforce this usage. Java makes no attempt at such separation – a class combines a module's specification and its implementation – and a supplemental tool such as JavaDoc must be used to separate these concerns.

Semantics for scalar data

One of Ada's and SPARK's strengths is the ability to specify ranges on scalar data, for example constraining an integer variable to be in the range 1 through 100. Whether it is used in testing during unit debugging, in data validation of user input during program execution, or in formal correctness proofs during system development, the specification of allowable ranges is an important Ada and SPARK idiom, absent from the C family of languages and Java. Range checking is automatically performed on array indexing, helping Ada and SPARK avoid the "buffer overflow" problems that are a well-known vulnerability in C and C++ programs.

Listing 1: Ada package structure

```
-- Package specification:
package Crypto is
   function Encode (Plaintext  : String) return String;
   function Decode (Ciphertext : String) return String;
end Crypto;


-- Package body (implementation):
package body Crypto is

   ... -- Auxiliary declarations

   function Encode (Plaintext : String) return String is
   begin
      ... -- Algorithm
   end Encode;

   function Decode (Ciphertext : String) return String is
   begin
      ... -- Algorithm
   end Decode;
end Crypto;
```

The package body for Crypto contains all of the implementation details for the Encode and Decode functions. Users of the package see only the source code for the package specification; the package body can be completely hidden.

Integer overflow is another potential source of insecurity. In Ada, this issue is addressed with well-defined and intuitive semantics: An integer overflow raises an exception rather than yielding an undefined result (as in C and C++, which presents a security vulnerability) or silently wrapping around to the most negative number (as in Java, which is unintuitive and error prone). SPARK goes even further than Ada, with static analyses to prove that an overflow could not occur for any input data.

Concurrency

Many kinds of applications are naturally concurrent, comprising a set of activities (referred to as "threads" or "tasks," depending on the specific language) that operate either with actual parallelism on a multiprocessor or are multiplexed on a single processor under the control of an operating system or runtime executive. Designing concurrent programs and demonstrating their correctness is intrinsically more difficult than for sequential programs, since concurrency introduces opportunities for new sorts of errors that can lead to bugs or insecurities:

- Race conditions, where the result of a program depends on the relative speed at which threads execute
- Data corruption, where one thread is assigning to a data object while some other thread is trying to access the same object
- Deadlock, where several threads are blocked and unable to proceed because each is waiting for some resource that is held by one of the other threads
- Starvation, where a thread is eligible to run but makes no progress because of preemption by other threads

Although such errors cannot be completely prevented through language features, Ada's concurrency model makes these problems easier to avoid. As one example, the semantics of Ada's protected object construct ensures that between the time a task checks the state of a shared object and when it takes some action based on that value, the state could not have changed. This assurance helps prevent a subtle race condition that could lead to an insecurity. SPARK includes a restricted set of Ada

concurrency features, known as the *Ravenscar Profile*[3], that is amenable to formal analysis. C and C++ do not have concurrency features, so the programmer needs to deal with these issues in a platform-specific manner based on whichever thread library is being used. Java's concurrency model is built on low-level primitives that are rather error prone and require extreme care in practice to avoid these problems.

Subsetting

Like their safety-critical counterparts, security-critical systems have a frustrating dilemma: Languages that might be useful because of their expressiveness are too complex to be used in their full generality. Their semantic richness is beyond the state of the art in demonstrating correctness, both for a program using the features and for the runtime support libraries that implement them. The solution in practice is to define subsets that are large enough to provide the needed functionality but small enough to allow verification of correctness. Indeed, the subsetting approach is being used for C, C++, Ada, and Java.

Two characteristics make Ada distinctive. First, its design is more orthogonal and thus more amenable to subsetting than other languages. As examples, MISRA C[4] has a number of ill-defined and awkward boundaries, and any attempt to subset Java by removing its dynamic flexibility immediately clashes with that language's main design philosophy. Second, through the language feature known as ***pragma*** *Restrictions,* Ada subsets can be defined by the application programmer, most likely on a project-by-project basis. As an example, it is straightforward to define a subset in which heap allocation is prohibited:

```
pragma Restrictions (No_Allocators);
pragma Restrictions (No_Implicit_Heap_Allocations);\
```

The specific subset can be tailored to the requirements of the application and to the analysis and verification techniques that will be used during development. This is preferable to using a "one-size-fits-all" subset defined by some third party, which might not match project requirements.

Formal methods and SPARK

In the safety-critical field, traditional testing-based verification approaches have been rather successful. However, it is well known that testing can reveal the presence of bugs but cannot demonstrate their absence. For security-critical systems, testing alone is insufficient; a higher degree of assurance is required. This can be accomplished through a more rigorous process: specify in advance the properties required of the program, and then formally demonstrate that these properties are satisfied by the running program. Achieving this type of assurance affects the entire process from start to end. It requires a language that allows the design decisions and requirements to be embedded into the code from the outset, and an associated toolset that can verify that these design decisions are met.

Ada can serve as an effective basis for such a language, but it would need to be restricted to avoid constructs that interfere with verification as well as augmented to include constructs that allow the specification of program properties in the source code. The SPARK language[2], developed by Praxis High Integrity Systems, may be regarded as the result of carrying out such a design. It inherits the advantages of Ada discussed previously but then adds a layer of "annotations," expressed in Ada comment syntax and illustrated in Listing 2, which allow a program to specify the criteria that must be satisfied by the running code. Static analysis tools process the annotations and either confirm that the program will meet its stated criteria or else detect and identify the defects. Detected insecurities include references to uninitialized variables, buffer overflow, storage overrun, arithmetic overflow, and division by zero. SPARK uses theorem-proving technology to show that a program could *never* exhibit such a failure at runtime for *any* input data. This offers a level of protection and assurance beyond any other language or toolset. The SPARK annotations are also designed so that the analyses are modular: You can analyze individual components and do not need to wait until you've finished the whole program to get meaningful results.

Listing 2: SPARK example

```
function Sqrt (X : in Natural) return Natural;
 --# return Y => (Y * Y) <= X and
 --#              (Y + 1) * (Y + 1) > X;
```

This example shows the specification of a function that returns the truncated integer square root of its argument. Note that Natural is the non-negative subrange of Integer values.

Watch your language: You never know who might be listening

For security-critical systems, and as summarized in Table 1, the language choice has a major impact on the ease or difficulty of demonstrating freedom from vulnerabilities.

Ada, with its emphasis on readability, its flexible modularization facility, its type model (including range checks on scalar data and mathematically intuitive overflow semantics), and its high-level concurrency features, is a stronger foundation for developing secure systems than C, C++, or Java. Moreover, Ada's flexible approach to subsetting lets applications define *a la carte* subsets that can reflect specific project requirements. SPARK, with Ada semantic underpinnings, goes further and allows documenting a program's intent through annotations and then generating formal proofs to verify that the program does what it is supposed to do and does not do what it is not supposed to do. Although building secure systems is a huge challenge, languages such as Ada and SPARK go a long way toward making this effort manageable.

Summary of language support for security requirements

	Ada	SPARK	C	C++	Java
Maintainability					
Ease of reading	High	High	Medium	Medium	Medium
Ease of writing	Medium	Medium	High	High	High
Modularity					
Spec/body separation	High	High	Low	Medium	Low
Namespace control	High	High	Low	Medium	Medium
Scalar data					
Range constraint	High	High	None	None	None
Index checks	High	High	None	None	High
Overflow checks	High	High	None	None	None
Concurrency support	High	Medium	None	None	Medium
Subsettability support	High	Not applicable	Low	Low	Low
Formal methods support	Medium	High	Low	Low	Medium

Table 1

Dr. Robert Dewar *is cofounder, president, and CEO of AdaCore. He has also served as a professor of computer science at the Courant Institute of New York University. He has been involved with the Ada programming language since its inception in the early 1980s and led the NYU team that developed the first validated Ada compiler. A principal architect of AdaCore's GNAT Ada technology, Dr. Dewar has written and presented on a variety of topics including software licensing, programming language issues, and safety certification.*

AdaCore
104 Fifth Ave., 15th Floor • New York, NY 10011
212-620-7300
dewar@adacore.com • www.adacore.com

Dr. Roderick Chapman *is a principal engineer with Praxis High Integrity Systems, specializing in the development of programming languages and static analysis tools for high-integrity systems. He holds a DPhil in Computer Science and is a chartered engineer and Fellow of the British Computer Society. Dr. Chapman is internationally renowned for his work on verification of correctness properties of high-integrity software.*

Praxis High Integrity Systems
20 Manvers Street • Bath BA1 1PX, U.K.
+44 1225 466991
rod.chapman@praxis-his.com • www.praxis-his.com

References

1. Ada Reference Manual, ISO/IEC 8652:200y(E) Ed. 3. www.adaic.org/standards/ada05.html

2. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, April 2003. ISBN 0-321-13616-0. See also www.sparkada.com

3. A. Burns, B. Dobbing, and G. Romanski. "The Ravenscar tasking profile for high integrity real-time programs," *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, Uppsala, Sweden, pp. 263–275. Springer Verlag, 1998.

4. The Motor Industry Software Reliability Association. *MISRA-C:2004, Guidelines for the use of the C language in critical systems*. MISRA Limited, 2004. See also www.misra-c.com

