Flight Software Programming Language Selection: A Security Perspective

Will Snavely[1], Craig Meyers[2], Chris Inacio[3], and Brad Runyon[4]
Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, 15213

The Department of Defense hoped to solve the "Programming Language Problem" once and for all with Ada. While Ada has experienced its fair share of success, it did not become the "one language to rule them all." This is in part due to the realization that language selection is a nuanced decision with many dimensions, including cost, schedule, staff experience, hardware, and—the focus of this paper—security. We propose a high-level approach for evaluating programming language security based on an international standard (ISO/IEC 24772:2013). Our discussion is focused specifically on flight software for space vehicles. We present experiences with a real flight software system, discussing how a better-defined process around language selection might have precluded certain threats to cyber-resilience.

I. Introduction

On April 29th, 1997, Assistant Secretary of Defense Emmet Paige Jr. issued a memorandum stating:*

> After carefully reviewing and considering the Board's final report, I have directed my staff to undertake the necessary actions to revise the policy contained in DoD Directive 3405.1, "Computer Programming Language Policy," to eliminate the mandatory requirement for use of the Ada programming language in favor of an engineering approach to selection of the language to be used.

Ten years prior, in 1987, the Department of Defense indeed had issued a directive that mandated the use of Ada—a programming language painstakingly designed and implemented at the DoD's behest—for new software projects [1]. Magnus Kempe, editor of the website *Ada Home*, explained in a 1997 article that the mandate failed not because Ada was a poor language—over those ten years Ada had been used in many important computer systems; rather, it failed in part because contractors viewed the mandate as being a hindrance to sound software engineering practice [2]. In particular: instead of having a programming language enforced, an engineering organization should have the agency to choose a language best-suited for a given task, based on sound technical evidence.

Later in his memo, Assistant Secretary Paige offered a rough outline of the factors that ought to be involved in the programming language decision process, including: performance, interoperability, reliability, safety, maintainability, available tooling, cost, schedule, staffing, and security. While each of these elements are worthy of discussion, there is a particular factor which is both critically important and historically underappreciated: security. Modern systems are often deployed in hostile environments—the Internet, for example—where adversaries will attempt to compromise them. Often, such adversaries will manipulate the system in ways the designers did not expect, leveraging vulnerabilities such as buffer overflows to threaten the confidentiality, integrity, and availability of the software. There are many strategies for improving the cyber-resilience of software systems; for example, the OWASP Software Assurance Maturity Model, illustrated in Figure 1, provides a variety of cyber-resilience best practices that can be applied across the software development lifecycle [3].

This paper explores the intersection between software assurance and programming language selection. To a first order of approximation, one could claim: insecure programs can be written in any programming language! While this is true, we believe that there is value in refining this concept, and understanding more precisely the security risk level associated with a given programming language choice. Additionally, language security risks are rarely absolute, and can be reduced or eliminated by applying external tools or practices. Therefore, a complete security assessment of a language should include an enumeration of risks along with the available strategies for mitigating those risks.

---

[1] Associate Software Engineer, CERT
[2] Principal Engineer, Software Solutions Division
[3] Technical Director, CERT
[4] Project Manager, CERT
* `http://www.adahome.com/articles/1997-04/po_memopaige.html`

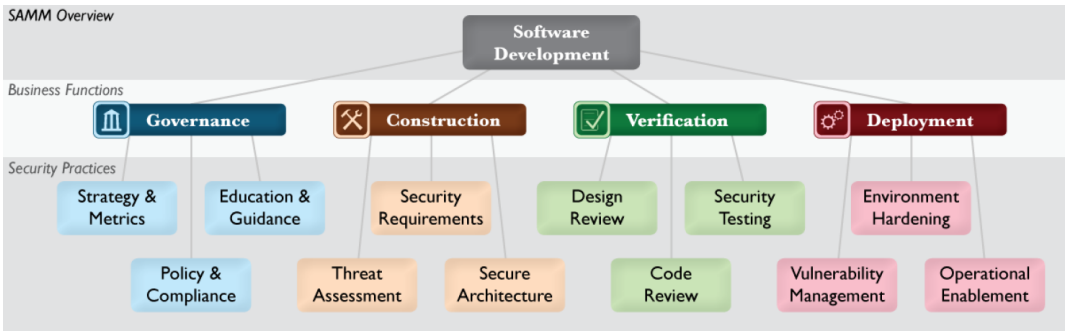

Fig. 1 An overview of the OWASP Software Assurance Maturity Model (SAMM).

Our discussion is further constrained by the domain of space vehicle flight software. Space vehicles, while physically quite remote, are not immune from cyber-threats [4]. Moreover, this is a domain where cyber-resilience is critical—compromises of these systems could seriously impact the DoD's mission. Finally, this constraint is useful in that it inherently reduces the set of languages to consider, given the (usual) real-time nature of these systems.

To summarize, we propose: when choosing a flight software programming language, one must be aware of the associated security risks, and of mitigations available to manage those risks. In particular, for a given flight software system, one must establish language risk level requirements—and in areas where the chosen language fails to satisfies those requirements, one must demonstrate that appropriate mitigations are in place. These requirements can vary from project to project, but for a space vehicle we expect these requirements will generally be very strict.

In this paper, we will present a notional language security assessment framework based on the international standard ISO/IEC 24772 (titled: "Guidance to avoiding vulnerabilities in programming languages through language selection and use"). We will briefly discuss how risk mitigations interact with this assessment framework, though the bulk of this subject is left as future work. We will not discuss a general approach for determining programming language security requirements—however, we will present a brief case study featuring a flight software system in Ada.

The structure of this paper follows. Section II gives a brief overview of the present-day programming language landscape for flight software. Section III presents a simple methodology for programming language risk assessment, and demonstrates application of this methodology to a small set of relevant languages. Section IV presents the aforementioned case study, regarding a flight software system in Ada. We provide concluding remarks in Section V.

II. Programming Language Survey

There are various resources that attempt to capture a snapshot of programming language popularity, for example the TIOBE index[†]. A shortcoming of such resources is that they are typically domain-agnostic. For example, PHP, Python, Javascript and SQL all are within the top 10 of the TIOBE ranking, as of August 2018—yet none of these languages are likely to be seen in an embedded system such as a satellite, for example because they require heavy-weight runtimes, or because they are optimized for other domains such as web applications. Of course, such languages would likely be present in the broader system surrounding a satellite, e.g. the ground segment. However, for the purposes of this discussion, we will narrow our focus to the flight software itself. In the following sections, we will evaluate the relevance of a variety of modern languages in this context.

C and C++: The Mainstream Monarchs

The "Department of Defense Software Factbook", published by the SEI in 2017, reports the following language breakdown for DoD systems: 63% C/C++, 22% Ada, 13% Java, and 2% other [5]. Nearly 70% of the projects analyzed as part of this effort (198 projects out of 287) were classified as real-time systems, which are typically embedded in nature. This provides reasonably strong evidence that C and C++ are dominant languages in the domain of embedded systems, for the DoD. It is not too much of a leap to extrapolate this result to flight software. Extrapolation aside, a 2009 NASA study on flight software identified C as the dominant language in this domain [6].

Michael Barr, in the first chapter of his book *Programming Embedded Systems in C and C++*, provides several

[†]`https://www.tiobe.com/tiobe-index/`

reasons for the immense popularity of C/C++ in this domain, including: these languages give programmers a large degree of direct hardware control, without being excessively low-level; many programmers have experience with one or both languages; they can be used on a wide variety of platforms/processors; compilers can emit very efficient, compact machine code; and so on [7].

Ada: Erstwhile Wonder Child

The introduction of this paper discussed a small episode in the history of Ada, namely, the inauguration and subsequent retraction of the DoD Ada mandate. We reiterate that this episode does not indicate the failure of Ada as a language. According to aforementioned SEI statistics, Ada is used by 22% of DoD software projects, a significant chunk of the market [5]. The SEI recently audited a DoD space system written in Ada, the subject of a case study later in this paper. Outside of the DoD, Ada has been used in many space vehicles. Dr. Michael B. Feldman, former chair of the SIGAda Education Working Group, maintains a list of software that uses Ada, which includes some thirty space vehicles from NASA, the European Space Agency, and other sources[‡]. While Ada is not as dominant as C/C++, evidence suggests that Ada is a mainstream language in the flight software domain.

However, there are some elements of the Ada ecosystem which are more emerging than mainstream. A prominent example of this is the SPARK variant of Ada. In the preface of their book, *Building High Integrity Applications with SPARK*, Peter Chapin and John McCormick describe the titular language as "a formally defined programming language and a set of verification tools to support the development of high integrity software [8]." In brief, SPARK allows one to write programs using a subset of Ada syntax, and then prove properties of those programs using a supplemental toolkit. There have been several academic research projects that have applied SPARK to the space domain, such as the Vermont Lunar CubeSat and its successor the Lunar IceCube [9], and David Lesen's Hi-Lite project [10]. In summary, there is strong evidence that SPARK can be successful in this domain.

Java: Making the Runtime Real-Time

While Java is certainly an enormously popular language (sitting at the top of the TIOBE index as of August 2018, for example), it is far less popular in the embedded/real-time domain. Java has features like garbage collection and a variety of implicit runtime checks that make it difficult to use in this domain. That said, there are various efforts to specify and implement variants of Java that serve this problem space, for example Embedded Java[§] and the Real-time Specification for Java (see [11] for a high-level description; moreover a new version of this specification is currently under development[¶]). A paper from Higuera-Toledano provides an overview of how these Java technologies can be applied to cyber-physical systems [12]. Overall, in the domain of flight software, Java is firmly in the emerging camp. Developers like Java, evidenced by its broad adoption in other industries, and as real-time/embedded support for the language continues to mature we may see greater adoption.

Rust and Friends: Making a Better C/C++

There have been various attempts to create improved versions of C and C++. Many of these attempts begin with C or C++, and add new features to shore up weaknesses in those languages, e.g. adding improved type systems or safer memory management. Examples include Cyclone [13], Deputy [14], and Checked C [15]. Most of these efforts haven't found much traction outside of academia, but nonetheless have made an impact on the next generation of systems languages.

Speaking of which: we have seen many new languages that aim to preserve the desirable features of C/C++ while mitigating the less desirable ones. Examples include Rust (from Mozilla [16][‖]), Swift (from Apple[**]), Golang (from Google[††]), and D (from Digital Mars[‡‡]).

While these languages have been successful, in no small part due their corporate sponsorship and support, there is little evidence that the aerospace industry is seriously considering this new generation of languages. There are likely many logistical issues with adoption, for example: the compilers for these languages may not have good support for the

[‡] https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html
[§] https://www.oracle.com/technetwork/java/embedded/overview/index.html
[¶] https://www.aicas.com/cms/en/rtsj
[‖] https://www.rust-lang.org
[**] https://developer.apple.com/swift/
[††] https://golang.org/
[‡‡] https://dlang.org/

hardware we are currently shooting into space. Despite this, the industry should devote resources to investigating these evolutions of C/C++.

Model-based Design

Model-based design (MBD) "emphasizes mathematical modeling to design, analyze, verify, and validate dynamic systems [17]." Instead of writing code directly in a language like C, one builds a formal model of a system, at an appropriate level of abstraction, and uses a toolkit to verify and validate this model. Depending on one's perspective, MDB represents either an evolution in software development, or a return to the mathematical roots of programming—the Dijkstra school of programming—made possible by improved tools and processes. Amazon has applied MBD to their cloud services, and have reported benefits such as: finding subtle bugs early; facilitating aggressive optimizations; and improving their overall design process [18]. To clarify, MBD does not necessarily preclude writing code in a traditional language. While some MBD methodologies involve automatically generating code from models, this is not universal. For example, Amazon did not generate code from their models; they still wrote code by hand. Some applications of MDB may involve a mix of generated code and human-written code.

The DoD has invested a significant amount of research into MBD, especially for cyber-physical systems, through programs like DARPA HACMS. A report on HACMS from Fischer discusses some of the reasons why MDB is in vogue, including: vast improvements in hardware resources; improvements in SAT solvers; a growing selection of mature toolkits (including Coq, Isabell, TLA+, Frama-c, and others); and mounting difficulty applying traditional software development practices to increasingly complex systems (especially high-integrity systems) [19].

Flight software in particular has been a popular target for MBD. The Orion Multi-Purpose Crew Vehicle, currently under development by NASA, implements MBD with Matlab, Simulink, and StateFlow [20]. Kerianne Gross, from AFRL, has released several publications over the past few years discussing the application of formal methods to attitude determination and control systems [21, 22]. There remain important challenges to address in this domain. Dr. Rob Petit, from the Aerospace Corporation, published a paper in 2013 discussing some of these challenges, including: ensuring consistency between models and code; making MBD methods scale to very large systems; and teams having to integrate multiple modeling languages together [23]. Regardless, there is broad agreement that MBD is a promising direction to move towards for complex, mission-critical systems, and work is ongoing to incorporate this methodology into modern software development lifecycles.

III. Language Security Risk Assessment

The fundamental question posed by this paper is: how does one assess the security risk introduced by choosing a given programming language? This section introduces a simple methodology for language security risk assessment, and applies that methodology to a set of popular programming languages. In particular, we will consider C, C++, and Ada, all of which are mainstream flight software languages. We additionally will consider SPARK and Java, two emerging languages in this domain.

We will first describe an abstract assessment model. We will then define a notional language security metric, by identifying a set of programming language vulnerabilities along with a simple means of scoring a language with respect to those vulnerabilities. We will then apply our scoring mechanism to the aforementioned languages. Finally, we briefly discuss how specific mitigations strategies might interact with this model, though this is largely left as an open problem.

A. Abstract Language Assessment Model

Figure 2 illustrates a high-level model for language security risk assessment. This model is built upon the notion of a security metric, represented here as an abstract numerical value. A system defines a requirement with respect to the security metric, for example a minimum acceptable value. Then, each programming language under consideration is scored on this metric. The baseline score for a language can be adjusted upwards, for example by applying mitigations, or downwards, for example by disabling runtime protections. The figure illustrates two scenarios: first, picking a language that does not meet the established security requirement, and having to apply mitigations to bring its score to an acceptable level; and second, picking a language that exceeds the security requirement, at the cost of high runtime overhead, and having to disable protections in order to improve performance (while still satisfying the security requirement).

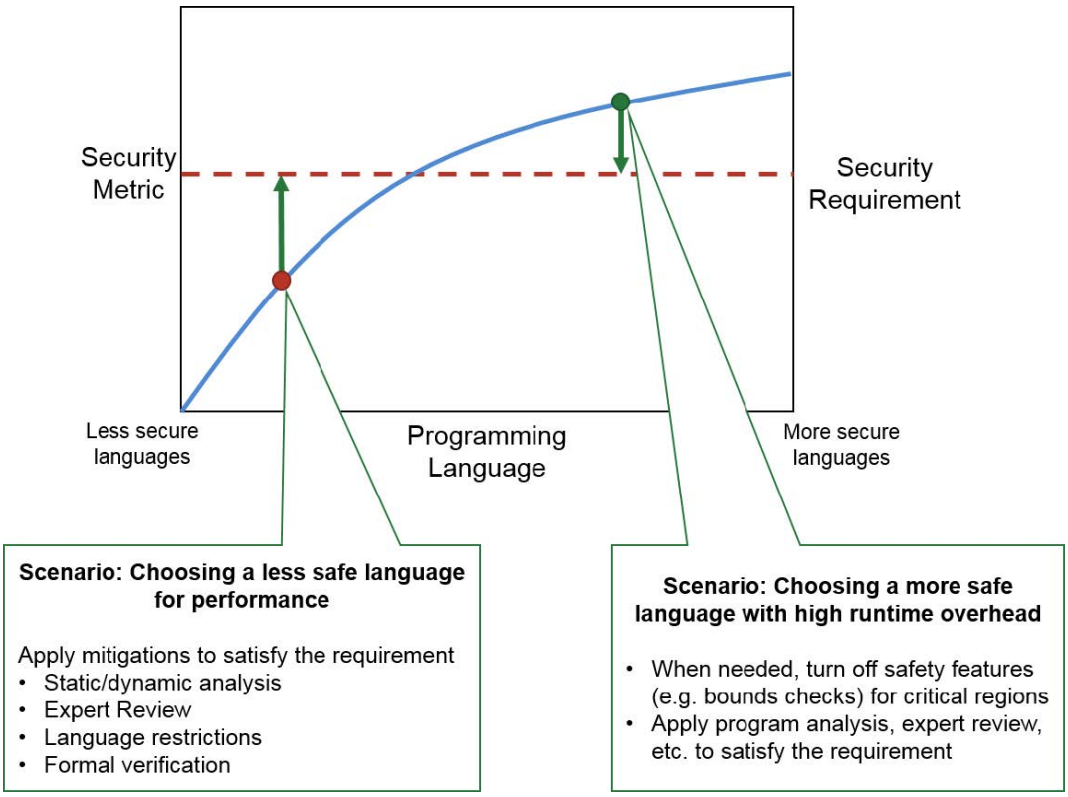

Fig. 2 A high-level language assessment model.

B. Identifying Programming Language Vulnerabilities

In order to define a language security metric, we must identify dimensions along which we can measure candidate languages. Generally speaking, we would like to enumerate the ways in which programming languages create security risks, and then assess the degree to which each language is susceptible to those risks.

There are a few existing software security weakness taxonomies for us to consider. The Common Weakness Enumeration (CWE), from the MITRE corporation[§§], is a very popular general-purpose software weakness taxonomy. The CWE has some nice properties for our context, for example the weaknesses in this taxonomy are generally language-agnostic. We can imagine selecting a subset of CWEs, and evaluating the extent to which a given programming language is susceptible to each.

Next, the SEI maintains secure coding rules for a variety of languages, including C, C++, Java, and Perl[¶¶]. Each language has its own curated set of rules. This is a desirable property in some contexts, e.g. establishing concrete coding guidelines for a program. For our purposes, having curated rules is less desirable, as we'd like to evaluate a wide variety of languages under the same criteria.

Finally, ISO/IEC TR 24772 is a standard that specifically targets the identification of vulnerabilities that affect programming languages. The standard identifies one of its applications as follows [24]:

> Developers of applications that have clear safety, security or mission-criticality are expected to be aware of the risks associated with their code and could use this Technical Report to ensure that their development practices address the issues presented by the chosen programming languages, for example by subsetting or providing coding guidelines.

This standard is therefore very well-aligned with our purposes. It strikes the right balance between the generality of the Common Weakness Enumeration and the programming-language focus of the SEI secure coding rules. ISO/IEC TR 24772 identifies 55 vulnerabilities that commonly affect programming languages, and moreover contains language-specific annexes describing the extent to which specific languages are affected by these vulnerabilities. Potential mitigations are also identified for each vulnerability. The 2013 edition of the standard has annexes for Ada, C, Python, Ruby, SPARK, and PHP, and it is likely new languages will be added in the future.

[§§] `https://cwe.mitre.org/`
[¶¶] `https://securecoding.cert.org`

For the purposes of illustration, we will focus on a subset of vulnerabilities from ISO/IEC TR 24772. It is likely a given engineering effort would need to engage in a similar scoping exercise, identifying the vulnerabilities most relevant to their context. The following listing describes the vulnerabilities we shall focus on from here forward. Each listed vulnerability specifies a three-letter unique identifier assigned by ISO/IEC TR 24772 (e.g. HCB for "Buffer Boundary Violation"). Note that these identifiers are not intended to be mnemonics, and were in fact arbitrary assigned by the standard. For each vulnerability, we provide a rough sense of severity.

1) **Buffer Boundary Violation (HCB):** Occurs when an array is read or written outside of its legal bounds. For example, if an array has 10 elements, and a read or write is issued at the eleventh element, an instance of this vulnerability has occurred. Violations are potentially very serious, in the worst cast permitting arbitrary code execution.
2) **Null Pointer Dereference (XYH):** Occurs when a pointer or reference with a `NULL` value is accessed as if it contained a valid address location. Violations typically impact availability, e.g. by causing program crashes, though in rare cases more serious consequences can arise.
3) **Arithmetic Wrap-around Error (FIF):** Occurs when a numeric value is incremented past its maximum value or decremented past its minimum value. For example, if an unsigned integer is decremented below 0, an instance of this vulnerability occurs. Violations can have minor or major consequences depending on how the wrapped value is used in a given program.
4) **Initialization of Variables (LAV):** Occurs when a variable is read before an appropriate value has been assigned to it. Violations can have minor or major consequences depending on how the uninitialized variable is used.
5) **Dangling References to Stack Frames (DCM):** Occurs when a variable allocated on a function stack frame is accessed after that stack frame is reclaimed. For example, if the address of a local function variable is returned by a function, and that address is subsequently read/written, an example of this vulnerability has occurred. Instances of this violation can result in read/writes to otherwise inaccessible regions of memory, which can have serious consequences, up to and including arbitrary code execution.
6) **Ignored Errors/Exceptions (OYB):** Occurs when an error or exception is possible in a given program, but that error/exception is not handled appropriately. Ignoring error conditions can result in application failure or misbehavior. This could be leveraged by an attacker to execute a denial-of-service attack, for example.
7) **Type-breaking Reinterpretation of Data (AMV):** Occurs when a value stored at a given location is reinterpreted in an invalid or unsafe manner. For example, if a program interprets an integer or floating point value as a memory address, then this vulnerability has occurred. Consequences range from minor to major depending on the manner in which the misinterpreted value is used.
8) **Memory Leak (XYL):** Occurs when software does not release allocated memory after finishing using it. If this happens repeatedly, large amounts of memory can be consumed, potentially causing an application to experience degraded performance or to crash. An attacker could leverage a memory leak as part of a denial-of-service attack.

C. Language Vulnerability Scoring

We illustrate a notional strategy for scoring a language with respect to the vulnerabilities identified above. For each vulnerability, a language is assigned a numeric value from 1 to 9, where 1 indicates high susceptibility to a vulnerability, and 9 represents little-to-no susceptibility. The rationale for a 9-point scale is so that values can be grouped into three rough categories: a score of 1-3 indicates "High" susceptibility, a score of 4-6 "Medium" susceptibility, and a score of 7-9 "Low" susceptibility.

As previously suggested, the baseline score a language receives for a given vulnerability is not absolute: the score may be adjustable upwards or downwards by various factors. For example, the application of mitigations for a vulnerability may adjust the score upwards. The suppression of runtime validity checks (e.g. justified by performance concerns) that prevent a given vulnerability may adjust the score downwards. In this scenario, if the score drops low enough, it may be necessary to introduce mitigations to readjust back upwards.

Figure 3 demonstrates an application of this scoring mechanism to five languages (C, C++, Ada, Java, and SPARK), using the eight ISO/IEC TR 24772 vulnerabilities identified above. Red cells indicate "High" susceptibility, yellow cells "Medium" susceptibility, and green "Low" susceptibility. This example scoring is not meant to be definitive, rather it is a rough approximation based on information in the ISO/IEC TR 24772 language annexes (when available) and author experience. The average language score is shown in the final row of the table.

On one end of the scoring spectrum, we have C and C++. Unsurprisingly, these languages introduce many potential security risks, virtually by design. For example, neither C nor C++ impose any requirements on compilers to check

ISO/IEC-24772 Vulnerability	C	C++	Ada	Java	SPARK
Buffer Boundary Violation (HCB)	2	4	8	9	9
Null Pointer Dereference (XYH)	2	2	8	6	9
Arithmetic Wrap-around Error (FIF)	2	2	8	6	9
Initialization of Variables (LAV)	3	5	6	7	9
Dangling References to Stack (DCM)	3	3	6	9	9
Ignored Errors/Exceptions (OYB)	3	3	6	6	9
Type-breaking Reinterpretation of Data (AMV)	3	3	8	7	9
Memory Leak (XYL)	2	2	5	8	9
Average	**2.5**	**3**	**6.9**	**7.3**	**9.0**

Fig. 3 Example language scoring against a subset of vulnerabilities.

whether array indices are inbounds. Moreover, programmers are responsible for managing their own memory (e.g. via `malloc`/`free`, `new`/`delete`). C in particular has a very loose type system, meaning variables can be fairly easily converted to other, potentially incompatible types. These factors and others create many possible opportunities for vulnerabilities.

On the other end of the spectrum we have Java and SPARK. SPARK in particular fares very well against this particular set of vulnerabilities, largely due to the fact that SPARK comes packaged with static analysis capabilities that can detect and prevent many common vulnerabilities. Java similarly features a runtime that detects and prevents common vulnerabilities.

Ada falls somewhere in the middle of the pack. Like Java, Ada has a runtime that prevents many flavors of vulnerabilities. However, Ada supports various means of circumventing these runtime safety checks, e.g. in pursuit of performance. For example, Ada features a set of "unchecked" operations—unchecked type conversion, unchecked memory deallocation— that expose programs to vulnerabilities similar to what one might experience with a C program.

D. Should One Always Choose SPARK?

This sample assessment might lead one to believe that SPARK is always the right language choice. Recall though that security is only one factor in the programming language selection process. While SPARK certainly has inherently better security properties compared to C, this does not mean it will always be the appropriate choice for a given flight software system. Factors like the target hardware and operating system might preclude the use of SPARK. Moreover, the developers responsible for a system might not have adequate experience with SPARK. These and other reasons might necessitate choosing a language like C or C++.

Additionally, choosing C or C++ does not necessarily spell doom and gloom for system security. These languages are perfectly suitable for critical systems, as long as the development efforts appropriately address their very real security risks. This cannot be taken lightly—for example, buffer overflow defects are extremely common in software (representing the majority of defects counted over a 25 year period by one study [25]), perhaps in part due to programmers using languages that are highly susceptible to buffer issues sans sufficient mitigations. A brief discussion of mitigations strategies can be found in the next section.

E. Language Risk Mitigations

There are many mechanisms for mitigating the risks introduced by a given language choice. The aforementioned OWASP Software Assurance Maturity Model [3] is one potential source of mitigation strategies, however the elements of this model are fairly broad, and may be difficult to map to individual language issues. Alternatively, ISO/IEC 24772

itself contains mitigation strategies for each identified vulnerability, both at the language-agnostic level and within the language-specific annexes. Mitigations that have particular relevance to programming languages include:

1) **Static analysis:** Static analysis inspects software for potential vulnerabilities without running it. Some languages, like SPARK, are very tightly coupled with static analysis by design. Other languages, like C, often rely on commercial-of-the-shelf tools to implement static analysis. Formal verification is a form of static analysis that involves mathematically verifying certain properties of software.
2) **Dynamic analysis:** Dynamic analysis involves searching for vulnerabilities while executing a given program. This might involve instrumenting a program to insert checks for unsafe behavior, e.g. instrumenting a C program to check for out-of-bounds array accesses, then running the program under various inputs to search for violations. There are many other techniques in this domain that are beyond the scope of this paper to discuss.
3) **Language subsetting:** One can opt to use a limited subset of a language in order to preclude the possibility of certain vulnerabilities. For example, one might forbid the use of dynamic memory allocation to avoid memory leaks, or forbid recursion to avoid overly-deep call stacks. Prominent examples of language subsets include: MISRA C [26] and the Ada Ravenscar profile [27].

There is a fundamental question of: for a given programming language choice, what mitigations should be applied in order to adequately manage the associated security risks? This is a complex issue with many dimensions, such as cost and schedule, and is an ongoing research subject. For example, in a 2017 SEI tech report, Alberts and Woody describe a "prototype software assurance framework" that attempts to synthesize cyber mitigations in a coherent, actionable fashion [28].

We certainly could approximate the impact of various mitigation strategies upon the vulnerability susceptibility scores presented above. For example, we might estimate that applying COTS static analysis tools to a C program may decrease its susceptibility to buffer overflows from "High" to "Medium", or that formal methods may further adjust the susceptibility to "Low". However, given the scope and complexity of this task, we leave this as future work.

IV. Case Study: Ada Flight Software

In mid-2017, the SEI assessed the cyber-resilience of a flight software system written in Ada. This system consisted of approximately 70K significant lines of code (excluding whitespace and comments). Initial findings from this assessment were discussed in a 2017 AIAA SPACE publication [4]. A key question we pursued with this assessment was: is the software free of buffer overflows (reads or writes outside of array bounds)?

In the general case, out-of-bounds array accesses in Ada are prevented by runtime checks. These checks raise exceptions upon failure. However, Ada permits developers to disable these runtime checks using compiler directives called *pragmas*. For example, there is a pragma called *Suppress_All*, which instructs the compiler to omit a wide variety of checks, including array bounds checks.

The flight software we assessed used the *Suppress_All* pragma in a number of locations, including modules which performed array accesses. The exact reason for specifying these pragmas is not known, but we might surmise that performance was one motivation. This raised the question: are any of the array accesses in the suppressed modules unsafe? Figure 4 illustrates this concern in the context of the language assessment model presented above.

A. Flight Software Static Analysis

In pursuit of evidence that the flight software was free from buffer overflows, we analyzed the code using a commercial static analysis tool for Ada. This tool returned over 4000 raw alerts, 1100 of which corresponded to potential out-of-bounds array accesses. A deduplication process reduced these 1100 alerts to 426. We ultimately inspected 44 of these 426 alerts, based on a simple prioritization scheme. 34 of these alerts appeared to be likely true positives—however, it is worth noting that all 34 of these issues originated from the same core defect in the software. Additionally, none of these potential out-of-bounds accesses occurred in a region with suppressed bounds checks. Overall, this static analysis did not give us strong confidence that the software had no buffer overflows, even though it did not identify any invalid array accesses in suppressed code modules. If out-of-bounds activity was overlooked in one place, it seemed probable that it might occur elsewhere.

B. Manual Code Review

We additionally performed a manual code review of modules with suppressed runtime checks, in search of potential buffer overflows. As a result, we identified one location where an out-of-bounds index appeared possible. The cause of

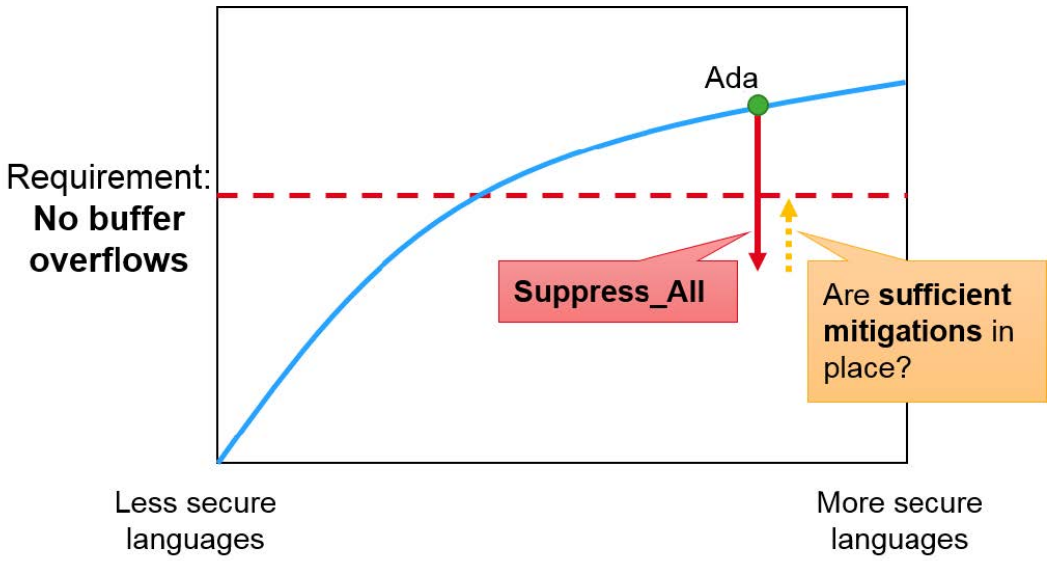

Fig. 4 Ada buffer overflow assessment.

this issue was an apparent typo in an array index calculation, which created scenarios where said index could hold a value larger than the length of the array.

C. Dynamic Testing

We have performed some initial testing with a simulator to try and recreate the candidate buffer overflows, discovered via automated and manual analysis, in the actual running software. Our success in this domain has been limited, so far. We were able to demonstrate that out-of-bounds accesses in suppressed regions indeed occur silently (no exception is raised). However we have not yet generated such accesses on the simulator via normal flight software execution. This does not mean it is impossible to produce this behavior: it might arise only under very specific circumstances which may be difficult to create with the simulator. We are continuing to investigate dynamic testing to assist with this task.

D. Results

Overall, our assessment of the Ada flight software was inconclusive with respect to establishing the absence of buffer overflows. Automated and manual static analysis identified a number of potential problems, though we faced difficulties recreating these issues in the running software. We believe that the developers of this software could have improved their process by explicitly recognizing and documenting the risk associated with suppressing runtime checks, and additionally providing evidence that this risk had been appropriately managed via mitigations like static analysis and manual code review.

V. Conclusion

The DoD did away with the "Ada mandate" in part because they realized the importance of making technically sound decisions with respect to language selection. To wit, the programming language selection process for mission critical systems must be aware of the security risks introduced by a given language choice, as well as the mitigations necessary to sufficiently manage those risks. In the paper, we have proposed a simple language assessment strategy based on ISO/IEC 24772, a standard identifying common programming language vulnerabilities. We don't claim that this specific evaluation strategy is ready for application to real systems, but hope that it serves to motivate structured decision-making processes for programming languages. We considered how such a process would have benefited a real flight software system inspected by the SEI, due to the presence of runtime check suppressions that potentially allowed buffer overflows. Considerable work remains to develop processes in this domain, for example accurately modeling the degree to which different mitigations (static analysis, dynamic analysis, code review) manage specific security risks.

Acknowledgments

The authors thank Dr. Roberta Ewart (Space and Missile Systems Center); and Dr. Wayne Wheeler, Nick Cohen, Jae Shin, VietHung Le, Dr. Joseph Betser, and Dr. Joseph Touch (all from the Aerospace Corporation) for their assistance with this work, for example giving us access to their simulator platform and patiently working with us to learn that system.

Terms

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

[1] Deputy Secretary of Defense, "Computer Programming Language Policy (DoD Directive 3405.1)," 1987.

[2] Kempe, M., "End of the Ada Mandate!" 1997. Retrieved from: `www.adahome.com/articles/1997-03/end_mandate.html`.

[3] Chandra, P., et al., "Software assurance maturity model," *A guide to building security into software development v1. 0*, 2009.

[4] Wheeler, W. A., Cohen, N., Betser, J., Meyers, C., Snavely, W., Chaki, S., Riley, M., and Runyon, B., "Cyber Resilient Flight Software for Spacecraft," *AIAA SPACE and Astronautics Forum and Exposition*, 2017, p. 5305.

[5] Clark, B., Miller, C., McCurley, J., Zubrow, D., Brown, R., and Zuccher, M., "Department of Defense Software Factbook," Tech. rep., Carnegie Mellon University, Software Engineering Institute, July 2017.

[6] Dvorak, D., "NASA study on flight software complexity," *AIAA Infotech@Aerospace Conference and AIAA Unmanned...Unlimited Conference*, 2009, p. 1882.

[7] Barr, M., *Programming embedded systems in C and C++*, O'Reilly Media, Inc., 1999.

[8] McCormick, J. W., and Chapin, P. C., *Building High Integrity Applications with SPARK*, Cambridge University Press, 2015.

[9] Brandon, C., and Chapin, P., "The use of SPARK in a complex spacecraft," *ACM SIGAda Ada Letters*, Vol. 36, No. 2, 2017, pp. 18–21.

[10] Lesens, D., Moy, Y., and Kanig, J., "Formal validation of aerospace software," *Proc. DASIA*, Vol. 11, 2013.

[11] Bollella, G., and Gosling, J., "The real-time specification for Java," *Computer*, Vol. 33, No. 6, 2000, pp. 47–54.

[12] Higuera-Toledano, M. T., "Java technologies for cyber-physical systems," *IEEE Transactions on Industrial Informatics*, Vol. 13, No. 2, 2017, pp. 680–687.

[13] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y., "Cyclone: A Safe Dialect of C." *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[14] Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. C., "Dependent types for low-level programming," *European Symposium on Programming*, Springer, 2007, pp. 520–535.

[15] Elliott, A. S., Ruef, A., Hicks, M., and Tarditi, D., "Checked C: Making C Safe By Extension," *Cybersecurity Development (SecDev)*, 2018. Submitted for publication.

[16] Matsakis, N. D., and Klock II, F. S., "The rust language," *ACM SIGAda Ada Letters*, Vol. 34, ACM, 2014, pp. 103–104.

[17] Jensen, J. C., Chang, D. H., and Lee, E. A., "A model-based design methodology for cyber-physical systems," *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, IEEE, 2011, pp. 1666–1671.

[18] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M., "How Amazon web services uses formal methods," *Communications of the ACM*, Vol. 58, No. 4, 2015, pp. 66–73.

[19] Fisher, K., Launchbury, J., and Richards, R., "The HACMS program: using formal methods to eliminate exploitable bugs," *Phil. Trans. R. Soc. A*, Vol. 375, No. 2104, 2017, p. 20150401.

[20] Tamblyn, S., Henry, J., and King, E., "A model-based design and testing approach for Orion GN&C flight software development," *Aerospace Conference, 2010 IEEE*, IEEE, 2010, pp. 1–12.

[21] Gross, K., Hoffman, J., Clark, M., Swenson, E., Cobb, R., Whalen, M., and Wagner, L., "Evaluation of formal methods tools applied to a 6u cubesat attitude control system," *AIAA SPACE 2015 Conference and Exposition*, 2015, p. 4529.

[22] Gross, K. H., Clark, M. A., Hoffman, J. A., Swenson, E. D., and Fifarek, A. W., "Run-time assurance and formal methods analysis nonlinear system applied to nonlinear system control," *Journal of Aerospace Information Systems*, Vol. 14, No. 4, 2017, pp. 232–246.

[23] Pettit, R. G., Mezcciani, N., and Fant, J., "On the needs and challenges of model-based engineering for spaceflight software systems," *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, IEEE, 2014, pp. 25–31.

[24] "Information technology – Programming languages – Guidance to avoiding vulnerabilities in programming languages through language selection and use," Standard, International Organization for Standardization, Geneva, CH, Mar. 2013.

[25] Younan, Y., "25 Years of Vulnerabilities: 1988–2012," 2012. Unpublished.

[26] Hatton, L., "Safer language subsets: an overview and a case history, MISRA C," *Information and Software Technology*, Vol. 46, No. 7, 2004, pp. 465–472.

[27] Burns, A., "The ravenscar profile," *ACM SIGAda Ada Letters*, Vol. 19, No. 4, 1999, pp. 49–52.

[28] Alberts, C., and Woody, C., "Prototype Software Assurance Framework (SAF): Introduction and Overview," Tech. rep., Carnegie Mellon University, Software Engineering Institute, 2017.