NPS-CS-04-006

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Proceedings of the
Center for National Software Studies
Workshop on Trustworthy Software

by

James B. Michael
Jeffrey M. Voas
Richard C. Linger

10 May 2004

Approved for public release; distribution is unlimited.

Prepared for: Center for National Software Studies

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RDML Patrick W. Dunne, USN
Superintendent

Richard S. Elster
Provost

This report was prepared for the Center for National Software Studies. Funding in support of the workshop and preparation of this report was provided by the Center for National Software Studies, Federal Aviation Administration, and Naval Postgraduate School. The workshop was conducted in cooperation with the IEEE Reliability Society.

Reproduction of all or part of this report is authorized.

This report was prepared by:

_____
James Bret Michael, Associate Professor
Department of Computer Science

Reviewed by:

_____
Peter J. Denning, Chairman and Professor
Department of Computer Science

Released by:

_____
Leonard A. Ferrari
Associate Provost and Dean of Research

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** 5/10/04	**3. REPORT TYPE AND DATES COVERED** Technical Report	
4. TITLE AND SUBTITLE: Proceedings of the Center for National Software Studies Workshop on Trustworthy Software			**5. FUNDING NUMBERS**
6. AUTHOR(S) James B. Michael, Jeffrey M. Voas, and Richard C. Linger			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-04-006
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, California 93943			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this technical report are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

Achieving what is often termed "software trustworthiness" has been an elusive goal, in part because of the lack of a clear understanding of why we sometimes succeed but often fail to produce trustworthy software. In order to better understand the challenges the United States faces in achieving the goal of producing and acquiring trustworthy software systems and products, a workshop on the subject of trustworthy software was held April 8-9, 2004 in Monterey, California. The results of the workshop are documented in this report and intended to be used as input to a national research and development (R&D) agenda on software trustworthiness.

14. SUBJECT TERMS **15.** Dependability, Trustworthiness, Software			**15. NUMBER OF PAGES** 72
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

Standard Form 298 (Rev.2-89)

THIS PAGE IS INTENTIONALLY LEFT BLANK

Proceedings of the
Center for National Software Studies
Workshop on Trustworthy Software

Naval Postgraduate School
Monterey, California
April 8-9, 2004

Sponsored by

Center for National Software Studies
Federal Aviation Administration
Naval Postgraduate School
IEEE Reliability Society

THIS PAGE IS INTENTIONALLY LEFT BLANK

Preface

Information technology, and software in particular, can have a significant influence on national security, business, and everyday living. As stakeholders in information technology, individuals and society collectively place varying levels of trust in information systems and the distinct products from which these systems are composed. The level of trust placed by an individual in a system or product is based on some combination of the perceived and actual qualities of the system or product.

Achieving what is often termed "software trustworthiness" has been an elusive goal, in part because of the lack of a clear understanding of why we sometimes succeed but often fail to produce trustworthy software. In order to better understand the challenges the United States faces in achieving the goal of producing and acquiring trustworthy software systems and products, the Center for National Software Studies, in cooperation with the Naval Postgraduate School, Federal Aviation Administration, and the IEEE Reliability Society, sponsored the Workshop on Trustworthy Software April 8-9, 2004 in Monterey, California. The results of the workshop are to be used at the National Software Summit—also sponsored by CNSS—and other forums for setting the national research and development (R&D) agenda on software trustworthiness.

The following challenges, posed in terms of the responsibilities of the software engineering community to produce trustworthy software, were raised during the workshop. The term "stakeholders" is used here to refer to any party that will judge the trustworthiness of a system or product. Examples of types of stakeholders are direct and indirect users, owners, developers, resellers, and independent evaluators.

- ***Improving our willingness to build software trustworthiness into systems and products from their conceptualization.*** Treating software trustworthiness as an afterthought, or placing an overemphasis on time-to-market, cost, and features, has made it difficult or sometimes impractical to retrofit systems and products with the properties (e.g., safety, reliability, and security) that stakeholders deem necessary for a system or product to be judged as sufficiently trustworthy. How do we instill such a philosophy into the developers and maintainers of software?
- ***Bettering our ability to both identify and characterize the attributes of a system that contribute or detract from software trustworthiness.*** There are competing definitions of "trustworthy software." Irregardless of which definitions we gravitate to, it is necessary to treat trustworthiness in a holistic fashion by identifying and characterizing the attributes that influence the opinion of a stakeholder about the level of trust to place in a system composed of individual interacting products comprising software, hardware, and "skinware." What are these attributes? How can we characterize the attributes so that they can be made visible to stakeholders?
- ***Providing stakeholders with both "practical" and "justifiable" levels of software trustworthiness.*** Attributes of a system that contribute to the confidence stakeholders place in the trustworthiness of a system or product can be difficult to measure. In addition, some of the desirable properties of systems and products, from the perspective of software trustworthiness, are undecidable, requiring us to rely on the use of approximation methods to achieve and measure them. We need

to explore the concept of "practical" trustworthiness, using a taxonomy of specific properties relevant for ensuring that particular hazards do not occur, and further classified according to the kinds of methods that are effective for establishing the properties. Further, it is not uncommon for properties to be interdependent, requiring tradeoffs to be made by relaxing or tightening one or more of the properties with the portfolio of properties of a system or product. We need to investigate how to effectively apply a taxonomy in conjunction with tradeoff analysis to obtain a justifiable level of software trustworthiness. Stakeholders have different expectations and operational profiles.

- ***Developing a means for assessing the trustworthiness of both families of products and systems-of-systems.*** Most modern systems are composed of distinct subsystems or components, each a system in its own right. It is possible to create many variants of the same product by modifying software—creating a family of products—in order to meet the needs of different customers. It is not possible to assess each product variant or subsystem in isolation of the products or systems with which they will interoperate. How does one assess the trust of such products and systems? In addition, a system-of-systems can be adaptable, providing for the "plug-and-play" (i.e., insertion and removal) of components. The plug-and-play capability is necessary to permit a system to be readily adapted for use in different contexts and modes of operation. Can we effectively assess software trustworthiness of these moving targets? Are there incongruities between traditional methods for assessing the properties of a system and modern software engineering methods? To what extent should we strive to develop automated and efficient methods for checking the trustworthiness properties of systems and products?
- ***Producing empirical evidence to determine how specific software engineering processes, practices, and methods lead to the realization of trustworthy software.*** A lofty goal for the software engineering community is to produce "low-surprise" software. However, many of the processes, practices, and methods we use to develop and maintain software have not themselves been thoroughly investigated; their utility is touted on the basis of anecdotal evidence. We need empirical evidence of the extent to which specific processes, practices, and methods make a positive contribution towards generating trustworthy software. However, collecting such evidence may prove to be problematic, due to the fact that it is more difficult to directly measure software quality than to achieve it; that is, we are faced with the "culprit" problem of never knowing that we built a "perfect" system. However, the situation is not hopeless because there are symbolic means for measuring quality, even in the presence of high levels of unpredictable software behavior.
- ***Perfecting our ability to communicate with stakeholders about the topic of software trustworthiness.*** Consumers and evaluators of systems and products need to make an informed assessment of the trustworthiness of software. However, stakeholders vary in their understanding of their understanding of the principles, mechanics, and practices of computing. How do we provide stakeholders with a limited knowledge of software and computing with enough information to accurately assess the trustworthiness of the system or product? Even a certificate or

label may be overwhelming for some stakeholders. Is it possible to distill ratings of trustworthiness into something akin to miles per gallon or BTUs? Similarly, what channels of communication can we provide the consumers and evaluators of systems and products so that they can provide useful feedback to software engineers? Would it be useful to provide stakeholders with *curricula vitae* for systems and products, along with proof-carrying code? How can we foster increased sharing of information within the software engineering community about the successes and failures to provide trustworthy software? Some repositories of such information exist, but we as a profession have for some reason failed to improve our track record on developing trustworthy software.

- ***Investigating the feasibility of creating independent organizations to evaluate software trustworthiness.*** We need to improve the believability, in addition to the transparency, of information about the trustworthiness of software. Independent assessment organizations can provide such information, but their ability to conduct this function will only be as good as the data which they can cull from the software engineer. If software engineers do not consider trustworthiness from the conceptualization of the product, then tracing properties and hazards back to the various levels of system and product artifacts will require some level of reverse engineering of the systems and products. We need to investigate issues such as the following: Who will pay for the cost of evaluation? How often should evaluations of a system or product be conducted? How does one evaluate the software contained a system-of-systems, in which software development may be provided by multiple competing contractors, suppliers of non-developmental items, and system integrators, all of whom want to protect the proprietary aspects of their software and the processes, practices, and methods that they use? Can we extend existing assurance frameworks, such as the Common Criteria, to assess the trustworthiness of systems and products?
- ***Enhancing our ability to encourage organizations to produce and maintain trustworthy software.*** Although there are many reports in the news media about stakeholder dissatisfaction with the trustworthiness of software, the software engineering community in general tends to project the image of itself as practicing a *craft* rather than being a *bona fide* profession. Apparently organizations require more than just marketplace incentives in order to spur them into producing and maintaining trustworthy software. What type of incentives (e.g., legal or economic) should be used and at what level (e.g., management, engineering, or both) should they be targeted? Should we try to measure the performance of manager or engineer against safety and security, rather than schedule and cost? Even if one looks at cost, it must be on the total lifecycle cost (as opposed to just the development cost) to achieve a sustained level of software trustworthiness.
- ***Reforming the procurement process.*** We could reform the processes by which the private and public sectors procure systems and products that must be trustworthy. This could include, but not be limited to educating procurement personnel on software systems and trustworthiness properties, developing system capability models and case studies for trustworthy software contracting, integrating education into DSMC curriculum, and incorporating academic personnel into acquisition organizations. Acquisition personnel may know that achieving high lev-

els of assurance is costly, but not realize that bearing the cost upfront is a good investment when viewed from a total-lifecycle perspective.

- ***Increasing our investment in education and the big "R" of R&D to improve on our track record of technology transfer.*** There needs to be more interchange between academia and industry—in both directions—along with research funding to support such collaborations. A first step would be to incorporate a large-scale system perspective into software engineering curricula, with the aim of establishing an interdisciplinary approach, focusing on embedded and network-centric systems, identifying trustworthiness considerations in integration of software and containing systems, and accommodating dynamic integration and reconfiguration of software and systems. Technology transfer should be encouraged by funding basic research, in addition to sponsoring technology transfer of state of art into state of practice: substantial R&D is required to extend theory into engineering practice. We also need to reduce the risk of technology adoption through demonstration projects to show scalability.
- ***Investigating the effect on trustworthiness of systems and products over a range of market conditions.*** Markets can consist of a single or multiple buyers: there is evidence that monopsonies can lead to the production of high-trustworthy systems and products. Does the same hold true of for duopsonies and oligopolies? What contributes to success in these restricted markets, while we often fail to provide trustworthy software for use in systems and products for the mass market?

Each breakout group addressed on the following frameworks: legal, technology, and market. There was insufficient time during the two-day workshop to refine the list of challenges—especially the controversial ones—into a form palatable by the majority of the participants. As a group we decided not to publish the unedited slides, leaving this as an exercise to be conducted by the CNSS and participants, for further exploration at the National Software Summit and other forums.

What can be said about the products of the breakout groups is that there was general agreement that each challenge must have associated with it both a description of the opportunity and one or more actionable recommendations. These in turn need to be refined into the appropriate form for consumption by policymakers, corporate managers and leaders, and others in decision-making capacities. Here is an example of one of the challenges the legal-framework group explored:

- Problem: Government is not adequately focused on trustworthy software (software-intensive products and systems)
- Opportunity: Government could leverage its buying power (~$60B in IT) to influence the trustworthiness of software
- Recommendations: (i) Improve specificity of trustworthiness requirements for use by the Federal Government as the acquirer of software; (ii) strengthen OMB A-11 (and similar rules for acquiring IT) and the FARs; and (iii) have NIST provide specific guidance for improving acquisition

To conclude, the participants of this workshop realize that they and the rest of the software engineering community have been at times tilting at windmills. Although the rate of our progress at being able to engineer rather than craft high-integrity, high-depend-

ability, or high-trustworthy systems and products has been slow, we have the hope and conviction to improve our lot and devise incentives to get past resistance within the community to move beyond the *status quo*. Some of the R&D challenges are likely of the same magnitude as those addressed in the Manhattan Project. In order to get the support of policymakers, we will need to help them sort out the implications and alternatives for conducting future projects.

THIS PAGE IS INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE IS INTENTIONALLY LEFT BLANK

Musings on Assurance and Trustworthiness in Cyberspace[1]

Marshall D. Abrams
The MITRE Corporation
abrams@mitre.org

This is not the first time I have addressed trustworthiness in cyberspace. I re-read "Striving for Correctness"[2] and found that there was little grounds for optimism. Little or no progress has occurred since that 1995 paper addressed simulation, testing, process modeling, structured programming, life-cycle modeling like the spiral model, use of CASE tools, use of formal methods, object-oriented design, and reuse of existing code. Reliance on these methods involves some element of belief since no validated metrics on the effectiveness of these methods exist.

One of our issues is to define Software Assurance or trustworthiness. Since Software Assurance is not a common term and does not appear in any glossary, I thought it useful to look at related authoritative definitions:

- **Adequate Security** — Security commensurate with the risk and the magnitude of harm resulting from the loss, misuse, or unauthorized access to or modification of information. [OMB Circular A-130]
- **Assurance** — Measure of confidence that the security features, practices, procedures, and architecture of an IS accurately mediates and enforces the security policy. [CNSS Instruction No. 4009, Revised May 2003]
- **Assurance** — Grounds for confidence that an entity meets its security objectives. [Common Criteria]
- **Information Assurance (IA)** — Measures that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and non-repudiation. These measures include providing for restoration of information systems by incorporating protection, detection, and reaction capabilities [CNSS Instruction No. 4009, Revised May 2003]

I propose the following definition:

- **Software Assurance** — Grounds for confidence that a software entity (e.g., program, process, module) meets its specifications.

A critical IT system must do exactly what is identified in its specification and not do anything that is not so specified. Correctness of software always has to be *with respect to a specification.* The fidelity of the written specification to the actual intentions of the user and acquirer (not always the same) is a major contributor to trustworthiness. Informally, assurance is a "warm fuzzy feeling" that the system can be relied upon to reduce residual risk to the predetermined level.

All current methods contributing to trustworthiness have shortcomings that make it impossible to establish assurance beyond reasonable doubt. That is, establishing correctness becomes a matter of belief, not proof. "Proof" is not what we taught on first exposure to algebra, geometry and cal-

[1] This essay does not necessarily represent the position of any organization, notably The MITRE Corporation or the Department of Homeland Security.

[2] Abrams, M.D. and M. V. Zelkowitz, "Striving for Correctness," *Computers & Security*, vol. 14, no 7, pp 719 - 738, December 1995

culus. Proof can be likened to a social process[3]—it is only the test of time where no flaw has been discovered that builds our confidence in the ultimate truth of a theorem.

David Parnas, among others, has suggested that an "assurance tripod" is required[4]: the combination of rigorous testing, evaluation of the process and personnel used to develop the system, and a thorough review and analysis of various products produced during development as a way to minimize risk.

The RADC Structured Programming Series[5] provided the foundations of software quality that appear to be honored more in the breach than in the observance. In the pragmatic end, managerial judgment determines resource allocation to correctness and assurance. Today's paradigm focuses on short-term aspects such as minimizing the time to market instead of trustworthiness. It appears that we know, or think we know, how to produce trustworthy software, but that we don't do it! I will refrain from discussing why.

Comparing the trustworthiness of software in 1995 and 2004, I conclude that the problem is not amiable to a purely technical solution. I believe that regulation and legislation are needed. In 2004 trustworthiness in software is a leap of faith as much as that which stimulated writing "the race is not to the swift, nor the battle to the strong…."[6]

[3] DeMillo, R., R. Lipton and A. Perlis, May, 1979, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM*, Vol. 22, No. 5, pp. 271-280.

[4] Parnas, D. L., A. John van Schouwen, and Shu Po Kwan, June 1990, "Evaluation of Safety-Critical Software," *Communications of the ACM*.

[5] RADC, 1974 – 1975, *Structured Programming Series*, RADC-TR-74-300 Vol I - XV

[6] Ecclesiastes (l. IX, 11)

Trust Base Metadata for CV-Carrying Code

William W. Agresti
Johns Hopkins University
agresti@jhu.edu

To define trustworthy software, we can draw upon conventional notions of trust in other contexts. Trust is the confident reliance by one party on the behavior of another party. This definition carries critical implications when it is adapted for use with trustworthy software:

- Trust is a relationship.
- It involves two parties -- not one.
- It involves behavior -- of the software.
- It involves expected behavior -- by the interacting entity.
- It is always conditional -- on the context and operational environment that obtains at an instant in time.
- It involves specific behavior occurring and other behavior not occurring. (NRC, 1999, p.13)
- By trusting, one party makes a commitment to the other, and with various degrees of conscious awareness, may be accepting risks to get benefits. (Denning, 2004)

The net effect of these considerations is that it is insufficient to say, "I trust this software." It is only reasonable to say, "I trust this software to do X and not do Y under conditions Z at time T."

People may want to know key information about any software before making commitments to use it. Information that might provide a basis for trust should routinely accompany software to provide something of a running story of its life, literally its *curriculum vita* (CV). Why shouldn't we be able to "Right-Click" or pull down the "About" menu and find information about why we should trust this software? When software updates are delivered via the Internet, deliver updates to this trust base also.

As a preliminary look at what might comprise this trust base, we would desire evidence from diverse major categories of –

- Process – What process was used to build and test it?
- People – Who designed it, built it, tested it, deployed it, used it?
- Structural – What architectural paradigms and assurance techniques were used? Were "building codes" followed? What assumptions were made about availability and reliability, and how were they met in the design?
- Execution/practice – Used in the past for what purposes? In what environments? With what outcomes?
- Independent Sources – Who vouches for what is in the other categories

The categories of information are not of equal value. Pointedly, there is no substitute for the code itself, and the next best thing may be information about its execution history. (Although, as we are reminded from the world of investments, "Past performance is not necessarily indicative of future returns.") Before there is much operational experience, information on process and structure may be helpful. Information about people can help: for certain software, why shouldn't the key architects be identified, with users having some opportunity (e.g., via a downloadable video) to understand their design rationale and their strategies for robust operation? The association of individuals with software products is consistent with concurrent trends in our community for IT

professionals to take personal responsibility for their behavior, as do professional engineers for their designs.

A natural concern is the potential for an overwhelming amount of such information, beginning with the original testing inputs and results. (Voas, 2004) So it is impractical simply to append volumes of data to the software and expect people to find what they need. Technologies can help with this problem. Bayesian belief networks may be particularly useful because they can reflect changing levels of support as more becomes known about the basis for trust. Semantic web efforts have produced Resource Description Frameworks (RDFs) and the Web Ontology Language (OWL), which are being used beneficially in a wide range of domains. Perhaps our trustworthy software community could make use of these advances. By capturing the trust information with structured metadata, it can be effectively searched to find relevant information.

Other communities have come together to define metadata structures. One current use of RDF is encouraging for an application to trustworthy software. RDF/XML provides the Gene Ontology Consortium with a controlled vocabulary with which to annotate gene data to support associations, such as

- "ISS means 'inferred from sequence similarity [with <database:sequence_id>]'
- IDA means 'inferred from direct assay'
- TAS means 'traceable author statement'" (W3C, 2004)

The similarities of the Gene Ontology metadata to trustworthy software are –

- It is dynamic so metadata evolves as new research uncovers relationships among genes and proteins
- RDF formats describe various kinds of evidence to support annotations, leading to "come from" graph structures
- Attribution and querying can be conducted at varying degrees of granularity and over non-uniform data sources
- It is not intended for the web – it just takes advantage of associated technologies

Metadata on trustworthiness would include information on how the software is being used, what parts of it are being used, and with what success, so others (people and other software and systems) have a chance to reason about whether the software should be trusted for their particular uses. This call for evolving metadata emphasizes the need for continuing update and modification and is analogous to other developments. Software black box recorders are now available commercially to capture data about the software's experiences executing within embedded systems (Thane, 2003). A software assurance case records critical information about the evidence, claims, and reasoning associated with critical aspects of software, with the "evolving" feature echoed in the recognized need to maintain the assurance case over time. (DNS, 2004) A key feature of a proposed software certification process is the collection of field data from user's environments. (Voas, 2000)

So, by analogy with proof-carrying code, perhaps software with its trust base CV integrated, and updated with evolving metadata over time, may be a useful concept. Our community should consider using available technologies to define metadata structures that facilitate reasoning about a crosscutting and critical property of software – its trustworthiness.

References

(Denning, 2004) Peter Denning, Introductory Comments, Workshop on Trustworthy Software, Naval Postgraduate School, April 8, 2004.

(DNS, 2004) Workshop on Assurance Cases, International Conference on Dependable Networks and Systems, DNS-2004, http://aitc.aitcnet.org/AssuranceCases/
(NRC, 1999) *Trust in Cyberspace*, Committee on Information Systems Trustworthiness, National Research Council, National Academy Press, 1999.
(Thane, 2003) Henrik Thane, "Time machines and black box recorders for embedded systems software," *ERCIM News* No. 52, European Research Consortium for Informatics and Mathematics, January 2003, 32-33.
(Voas, 2000) Jeffrey M. Voas, "Developing a usage-based software certification process," IEEE *Computer 33*, August 2000, 32-37.
(Voas, 2004) Jeffrey M. Voas, Discussion Comment, Workshop on Trustworthy Software, Naval Postgraduate School, April 9, 2004.
(W3C, 2004) W3C, *RDF Primer*, Downloaded from W3C Web Site, http://www.w3.org/TR/rdf-primer/, April 5, 2004.

Trustworthy Software Systems

Larry Bernstein
Stevens Institute of Technology
lbernstein@ieee.org

Software system development is too often focused solely on schedule and cost. Sometimes performance and functional technical requirements become an issue. Rarely is trustworthiness considered. Not only must software designers consider how the software will perform they must account for consequences of failures. Trustworthiness encompasses this concern.

Trustworthiness is a holistic property, encompassing security, safety and reliability. It is not sufficient to address only one or two of these diverse, dimensions, nor is it sufficient to simply assemble components that are themselves trustworthy. Integrating the components and understanding how the trustworthiness dimensions interact is a challenge. Because of the increasing complexity and scope of software, its trustworthiness will become a dominant issue.

Software fault tolerance is at the heart of the building trustworthy software. Microsoft claims to have undertaken a Trustworthy Computing initiative. Bill Gates sent a memo to his entire workforce demanding, "… company wide emphasis on developing high-quality code that is available, reliable and secure- even if it comes at the expense of adding new features." Information Week, Jan. 21, 2002, issue 873, p.28.

Trustworthy software is stable software. It is sufficiently fault-tolerant that it does not crash at minor flaws and will shut down in an orderly way in the face of major trauma. Trustworthy software does what it is supposed to do and can repeat that action time after time, always producing the same kind of output from the same kind of input. The National Institute of Standards and Technology (NIST) defines trustworthiness as "software that can and must be trusted to work dependably in some critical function, and failure to do so may have catastrophic results, such as serious injury, lost of life or property, business failure or breach of security. Some examples include software used in safety systems of nuclear power plants, transportation systems, medical devices, electronic banking, automatic manufacturing, and military systems.

Modern society depends on large-scale software systems of astonishing complexity. Because the consequences of failure in such systems are so high, it is vital that they exhibit trustworthy behavior. Much effort has been expended in methods for reliability, safety, and security analysis, as well as in methods to design, implement, test, and evaluate these systems. Yet the "best practice" results of this work are often not used in system development. A program is needed to integrate these methods within a trustworthiness framework, and to understand how best to ensure that they are applied in critical system development. In addition, it is important to focus attention on critical systems and to understand the societal and economic implications of potential failures. There must be consequences to those who continue to deliver failure prone software.

Much software engineering focuses only on features and schedule, especially schedule. My view is that a shift is needed. The software engineer must make judgments or tradeoffs among the functions the software provides, the time it will take to produce the software, the cost of producing the software, how easy it is to use and how reliable it is. A reliability based focus is needed. The fundamental software reliability equation is:

$$\text{Reliability} = e^{-k\lambda t},$$

where k is a normalizing constant,

λ = Complexity/ effectiveness x staffing,
and t is the time the software executes from its launch

This equation can be used to make engineering tradeoffs. It is reasonable, if unorthodox, to model the software engineering process based on this model. The longer the software executes the more likely it is to execute a latent fault that soon becomes a failures. Failures are hangs and crashes of a system. The term λ incorporates the factors a software project manager controls through the development process. By providing better software tools, such as higher level languages, to the software designer the reliability of the final product is better. By reusing reliable components and properly integrating them the software project manager reduces the complexity of the system again making it more reliable and by adding staff well beyond the minimum staff predicted by staffing models more effort can be placed on such activities as diabolic testing and system audits to make the system more trustworthy.

The ethical software engineer makes sure that a product solves the customer's problem, that it is tested, that good software engineering practices were used in its development and that any limitations of the product are clearly stated.

Ethical behavior can not be assured within the profession. There is no penalty for software engineers who are unethical according to the IEEE/ACM professional code:

ACM/IEEE-CS Joint Task Force on
Software Engineering Ethics and Professional Practices
Short Version
Preamble

> The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.
>
> Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:
>
> 1. PUBLIC - Software engineers shall act consistently with the public interest.
>
> 2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.

8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Too often software professionals do not think about the risks to others. And when they do, they are frequently overruled by their bosses or product managers. Laws are needed that require every product to have a named Software Architect and Software Project Manager. The same person may perform both roles. The roles are:

Software Architect:

1. Affirms that the software product solves the customer's problem
2. Affirms that the software product is suitably reliable, easy-to-use, extendible, not harmful and robust. That it is trustworthy.
3. Affirms that the requirements are valid.

Software Project Manager

1. Affirms that the software was successfully tested against the requirements.
2. Affirms and identifies the good software engineering processes were used in the software development and integration.
3. Affirms that the project is within budget, on-time and performs satisfactorily

Trustworthiness is already an issue in many vital systems, including those found in transportation, telecommunications, utilities, health care, and financial services. Any lack of trustworthiness in such systems can adversely impact large segments of society, as exemplified by recent outages of telephone and Internet systems, due in part to failures in software systems. It is difficult to estimate the considerable extent of loses experienced by individuals and companies that depend on these systems. The issue of system trustworthiness is not well known or understood by the public or the nation's leadership. The societal impact of failures in critical systems must be understood.

Trustworthiness as Risk Abatement

Valdis Berzins
Naval Postgraduate School
berzins@nps.edu

1. What "Trustworthy Software" Means

In an ideal scientific world, trustworthy software would carry absolute guarantees that the software will perform its required functions under all possible circumstances, will do so on time, and will never perform any actions that have hazardous consequences.

Unfortunately, such absolute guarantees are impossible: even if the software (including the operating system, compilers, loaders, DLL's, etc.) is mathematically proven to have specified properties, those proofs are checked by computers, and the checking software is similarly proved and checked (using itself), there still exists the possibility that faults in the checking software could make it unable to detect some faults, including its own.[7]

Other sources of imperfection that are more significant in practice include: (1) faults in the formalization of the properties that need to be established, (2) possibly invalid hypotheses about the properties of the environment of the software, (3) possibly invalid hypotheses about the needs of the people the software is supposed to serve, and (4) changes in the environment that occurred after the software was constructed. Particularly for first-of-a-kind systems, a majority of system faults can be traced to requirements and specification errors, even in the context of the imperfect quality assurance processes typical of current commercial practice. Note that sources (2) and (3) cannot be eliminated by purely symbolic analysis, because they concern the connection between the software and the physical and social world. We conclude that some degree of uncertainty is inevitable; the remaining questions are how much uncertainty, what kinds, and how they can be effectively reduced.

In practical engineering situations costs and delivery schedules must be taken into account. A practical concept of trustworthiness should therefore consider the time and cost required to establish given measurable levels of confidence in specific properties. Many properties of potential interest are undecidable in the general case, and require approximations, huge amounts of computation, and/or expensive human guidance in practice. **The amount of effort and expense warranted to establish high levels of confidence in a property and the degree of confidence needed depend on the severity of the consequences of violating the property.** Cost considerations also put a premium on relevant properties that can be established at low cost, by effective non-interactive algorithms that are efficient enough to handle large problems in practical amounts of time [1], even if those properties are relatively simple and do not capture all aspects of trustworthiness.

Practical trustworthiness should therefore be separated into a taxonomy of specific properties relevant to ensuring that particular hazards do not occur, and further classified according to the kinds of methods that are effective for establishing the properties.

[7] This process is commonly considered to be prohibitively expensive on a large scale, and also on a small scale if it has to be redone every time the software is modified.

2. How to Ensure that Commercial Software is of Higher Integrity

Software integrity has two parts: building the right system, and building the system right.

The first part is best addressed by improvements in requirements analysis, including improved support for prototyping, checking that software requirements and trustworthiness properties imply system level requirements and trustworthiness properties, establishing completeness of hazard analysis, and establishing confidence levels for preventing hazards of given severity levels. Sound systematic methods with measurable results are needed to eventually support standards and certification organizations.

The second part is best addressed by finding fully automatic efficient methods for certifying absence of particular kinds of hazards, by developing assured methods of synthesis that certify some properties of interest by virtue of disciplined and automatically checked construction, such as certified software generators for particular domains, and by developing methods to certify trustworthiness properties at the architecture level, to help maintain the trustworthiness of systems that must change, either quasi-statically or via dynamic adaptation.

Automated methods for checking trustworthiness properties can form the basis for practical certification standards that will ensure certified software is more trustworthy that current commercial software, even if they leave possibilities of other types of failure still open. Examples of such properties that should be automatically checkable in the relatively near term include:

1. All exceptions that can be raised are handled
2. The program cannot deadlock
3. The program will never read an uninitialized variable or unopened file
4. Every opened file is closed before the program terminates
5. The program will never dereference a null pointer or cause a segmentation fault
6. Parallel threads never simultaneously write/(read or write) the same location
7. Every lock is eventually released
8. Absence of memory leaks
9. Absence of buffer overruns
10. Inability of unauthorized users to access files or run programs

An additional challenge that should be solvable in the longer term is to statically certify that all programs that can be produced by a given software generator satisfy properties such as the above.

References

[1] V. Berzins, Lightweight Inference for Automation Efficiency, Science of Computer Programming 42 (2002) 61-74.

Safety Critical Software Development

Michael L. Brown
Naval Postgraduate School
mlbrown@nps.edu

Software, in one form or another, affects nearly every aspect of our lives. Embedded real-time systems in our home awaken us, make our morning coffee, heat our morning muffin, and control the temperature of our house. Few people realize the extent to which software affects our lives. We drive to work in automobiles with dozens of embedded real-time computer systems: they meter the fuel to our engines, control interior and exterior lights, ensure that the engine is operating a maximum efficiency (or, at the flick of a switch, maximum performance), and they help prevent collisions by pulsing our brakes to prevent wheel lock-up should we have to brake suddenly. All of these embedded real-time systems have safety implications: should they fail in a specific ways, they may result in harm to the user, property, or the environment. Nowhere is the use of computers and software more prolific than in military systems. Virtually all weapon systems rely on computers and software to achieve their mission. The functionality of most modern weapons is entirely dependent on computers: without the computers and associated software, the weapon cannot perform any aspect of its mission. Not only is the functionality dependent on computers and software, but the safety of those systems is as well.

Processes for reducing the risk associated with software in safety-critical roles are well known and documented for individual systems. Within the confines of the system under development, known as the Engineering Design Space, the System Safety Engineers reduce the risk associated with the system by careful design selection and implementation of safety-related functions and other safety-related aspects of the system and software design. However, all of the Software Systems Safety processes require the System Safety Engineer have detailed insight into the design of the system and software, the operational use and operational environment of the system, and the system's interfaces to external systems. In some cases, the System Safety Engineer can influence the design of the interfacing systems (part of the operational environment) to mitigate risks that are outside the Engineering Design Space of the system under development to mitigate potential risks associated with commands and data across the interfaces. These recommendations, if implemented, become a critical aspect of the risk assessment and safety assurance for the system under development. These factors, including the concept of operations, define the system context in which the system under development will function. The risk associated with the system depends on this wider system configuration and operational environment. This is especially true when these interfaces include safety-related functionality and data.

Safety is a property of the entire system: the safety risk associated with any component of a system, whether a simple resistor or a complex algorithm, depends on its interface and interactions with the entire system. As customers demand that the Systems Engineering and especially the Software Engineering disciplines develop larger, more complex systems, the "components" of the system of necessity become larger and more complex. This leads both disciplines to build systems with reusable components rather than developing unique components to achieve the necessary functionality. The reuse of components, whether hardware or software, moves the practicing engineers to a higher level of abstraction, one in which the creative processes involve large components rather than detailed implementation. In the Software Engineering discipline, this trend is evident in the use of Object-Oriented Analysis and Design, Generative Programming and Aspect Oriented Programming. Each of these methodologies moves the Software Engineer farther from the details of the software and its host system.

Ideally, we would like a library of "safety-certified" components with which to build complex systems. However, as noted earlier, the safety of a component depends on many factors including its interaction with other components of the system, their potential failure modes both singly and in combination with other components, the operational use and environment, etc. The level of abstraction at which modern system development occurs does not provide the detail necessary to identify the safety concerns, especially since many of these aspects may not be known until run-time. Without foreknowledge of these safety-critical aspects, we cannot certify a component as safe.

The Software Systems Safety process, as defined in the Joint Services Software Systems Safety Handbook and many other sources, was designed to be compatible with traditional software and systems development methodologies such as Grand Design and Spiral Development. In these development paradigms, it is both practical and effective. However, its reliance on detailed knowledge of the interactions between system components and the system and its environment is not compatible with modern system and software development methodologies.

On-going research related to Software Systems Safety significantly lags behind research into software development methodologies. The Software Systems Safety discipline is just now beginning to address "safety contracts" for classes of objects developed under the OOA&D methodology. However, the successful application of safety contracts is yet to be demonstrated. Meanwhile, the software engineering discipline moves forward with the next generation of development methodologies.

Our increasing reliance on software to control potentially safety-critical systems requires that we initiate research into development, analysis, and assessment techniques that ensure that software developed for complex systems will function in the system context and operational environment with an acceptable level of safety risk. Modern software development methodologies require new paradigms for Software Systems Safety, paradigms that provide the same level of assurance that the current process provides. Some examples include:

- Formal process definitions that fully integrate Software Systems Safety into the Systems Engineering and Software Engineering processes[8]
- Formal semantics for Aspect Oriented Programming that can specify the safety attributes of software components, tools that bind the safety attributes to the software design through implementation, and tools that verify semantics hold through the system and software design and development, component selection and integration, and instantiation of software components at runtime.
- Formal semantics for specifying safety contracts in Object Oriented Design and Analysis along with tools that bind the safety contracts throughout the design and development process.
- Testing tools that verify the implementation of safety attributes in the resultant system.
- System of systems architectures that support desirable safety attributes.

This list identifies only a few of the critical needs in this discipline. Many other development methodologies will require different techniques for ensuring that the software resulting from the development process provides an acceptable level of safety risk in the system context.

[8] Proposed extensions to the Software Engineering Institute's Capability Maturity Model Integrated begin to address this issue but are inadequate in their current form.

Can We Engineer Dependable System-of-Systems Software?

Dale S. Caffall
Missile Defense Agency
butch.caffall@mda.osd.mil

Background

The Department of Defense (DoD) looks increasingly towards an interoperable and integrated system-of-systems to provide required military capability. While the need for networked capability has exploded in military warfare, DoD has yet to develop a successful acquisition methodology for acquiring system-of-systems that will yield dependable system-of-systems that provide enhanced military capability in the intended battlespace. Our current systems-of-systems tend to require a great deal of software maintenance and to be intolerant of even the most minor of changes with respect to system behavior. (N.B.: In this paper, we define a system-of-systems as an amalgamation of legacy systems and developing systems that provide an enhanced military capability greater than that of any of the individual systems within the system-of-systems.)

As examples of system-of-systems, consider the joint air defense system-of-systems environment and joint information transfer system-of-systems. DoD has invested many millions of dollars in providing effective, interoperable, and robust systems; however, these systems require significant software maintenance and numerous software patches to limp along in support of our warfighters. Many identified dependability and integration deficiencies have plagued these systems for years; however, due to the considerable software complexity of these systems, our approach to solving these problems is oftentimes the development of a software patch to suppress undesired system behavior. This oftentimes results in undesired system behavior in other subsystems of the system-of-systems.

In the current DoD environment of rapidly paced acquisitions, senior management tends to place intense pressure on delivering desired features in seemingly unrealistic timeframes. In this type of acquisition environment, systems may be developed with little thought about the organization and behavior of the software. By rushing to deliver something quickly, system developers maintain a vision as far as the next line of code or the next aspect of detailed design. This is more evident in the acquisition of a system-of-systems than in a single system

We propose that software engineers employ two single-system software-development methods in the development of system-of-systems software: software architectures and formal methods. While many other tools and procedures exist for software development, we believe that these two efforts would significantly increase the dependability of system-of-systems software. We will use a hypothetical missile defense system-of-systems with a battle manager for our case study in this paper to explain our beliefs.

System-of-Systems Software Architecture

Unlike other development and construction efforts, software developers oftentimes are seemingly quite content to design and develop software without a roadmap that depicts how software components are organized in a system, how these components fit together, how these components interact, and how these components fulfill the system requirements. Such a roadmap for software development is a software architecture that defines the system in terms of computational components and the interactions among those components. Additionally, the software architecture bridges the gap between system requirements and realization, thereby documenting the rationale

for design decisions. Although users will quickly know whether something is wrong when they select a menu item and the system crashes, software engineers frequently cannot identify if the problem is in the operating system, in the application, or in the network.

Oftentimes, we compare engineering disciplines to seek points of commonality. Designing and constructing a new building is a common metaphor from which we attempt to draw lessons learned from other engineering disciplines. In designing a skyscraper, a journeyman architect will elicit requirements from the client and translate those requirements into various views of the proposed structure. A civil engineer will develop structural drawings and construction plans to build the new structure. Without some type of framework from the architect, the civil engineer could not construct a skyscraper that would support its own weight as well as the weight of the occupants, and their associated furniture and work materials.

The accepted framework for skyscraper design and construction is the set of blueprints for the proposed building. This set of blueprints has many views to include the physical construction of the building, more detailed construction views of each floor, views of the heating and cooling systems, views of the plumbing, and so forth. In essence, the skyscraper's architect develops this set of blueprints to provide a conceptual framework of the proposed building that all stakeholders can read and understand.

Imagine constructing such a building without a framework. Suppose that our civil engineer was somehow able to construct the skyscraper to the height of three floors whereupon which he loses confidence in constructing additional floors. Consider that the civil engineer now wants to add electrical wiring, communications cables, heating ducts, water and waste drainage plumbing, gas lines, etc. Now, the various craftsmen must drill holes; run wiring, pipes, and ducts; and connect appliances to this skyscraper without the benefit of any visual framework. We can only imagine the hodgepodge of wires, cables, tubing, pipes, ducts, and appliances that would result in this construction.

It is important to emphasize that the set of blueprints is a collection of many views of the new building. Each view is important to the overall construction of the building. Without the full set of views, some portion of the new building will suffer in the construction phase.

While a software architecture will not guarantee that a system-of-systems meets its requirements, a poorly designed or ill-defined software architecture makes it nearly impossible for the software developers to realize a system-of-systems that meets its requirements. Typically, the system-of-systems architecture is little more than a "sticks-and-circles" diagram in which the circles represent the various systems in the system-of-systems and the sticks represent the communication links among the systems. More often than not, this type of architectural view represents the totality of a system architecture effort in DoD organizations. Unfortunately for the developers who require information models that faithfully represent the operational battlespace, the circles of the sticks-and-circles diagram do not define the behavior of the systems and the sticks reveal little of the connectors that these lines represent. The information model formed by the sticks-and-circles diagram is a weak information model.

Much too often, we initiate coding from a reasoning about the "sticks-and-circles" diagrams. During the development, we add new layers of features and functional enhancements to the system software without clear insight into the organization of the system software. Inevitably, the basic software organization that seemed so reasonable at the beginning begins to break apart under the weight of the system software revisions. Regrettably, the software development becomes another casualty to report in future studies as to why software developments are not successful.

We can see that the conceptual framework is essential for reasoning about the problem. In the absence of such a framework, we can easily predict the degree of success in the above endeavors. We can see that the multiple views add richness to our conceptual view of the problem and potential solutions.

While keeping in mind the idea that multiple views of a system can increase our understanding of system behavior, we will apply this concept to the system-of-systems problem. Consider the following hypothetical missile defense system-of-systems that we use to illustrate both the inadequacies of the "sticks-and-circles" system architecture view and the value of the software architecture views. Let us define a proposed missile defense system-of-systems as four sensors of differing type, four battle management systems with organic sensors of differing type, and four weapons launchers of differing type as depicted in the below diagram of Figure 1. The challenge to our system engineer is to integrate these twelve systems into a single system-of-systems.

In all likelihood, the program manager for each system never had a requirement for inclusion of that system into a system-of-systems environment. Thus, these twelve systems were developed in isolation from the other systems and each system design was developed in a different manner than the others. Most importantly to dependability, the software organization differs among the systems.

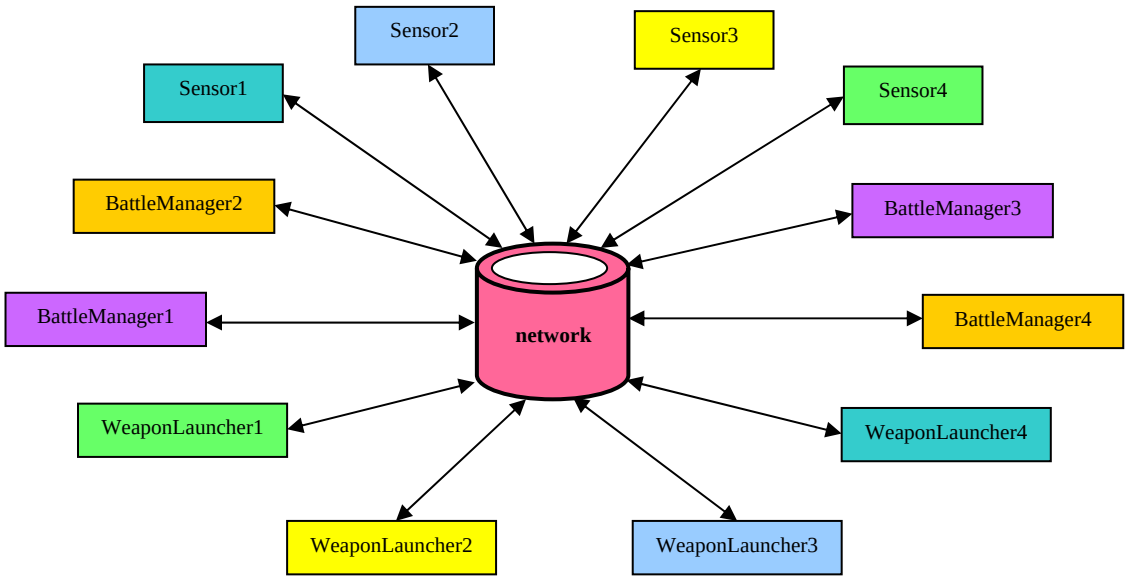

Figure 1. Hypothetical Missile Defense System-of-Systems

The traditional solution is to apply a communications solution for interoperability and integration. That is, the "stick" will be a means of information transfer, a messaging protocol, and, perhaps, a translator box to translate the messaging format from one system to another. Traditionally, this methodology has failed to achieve an interoperable and integrated system-of-systems. With each new failure, the system engineers attempt to "tighten up" the protocol standard; however, the system-of-systems did not achieve the desired degree of interoperability and integration. The

end-state was a collection of systems that are tightly coupled with a realized protocol standard that only served to increase the system-of-systems software complexity.

Because architectural decisions are usually made early in the lifecycle, these decisions are the hardest to change, and hence the most critical and far-reaching. Without a software architecture that faithfully models desired system behavior, it is difficult to achieve the satisfaction of the original performance and behavioral requirements, and it is probably impossible to accommodate major design changes. Software architectures serve as a planning tool for allocating system requirements as well as promote the construction of subsystems from architectural components – not the other way around. Furthermore, problems with the requirements and the architecture will result in design problems via refinement into lower-level system artifacts such as detailed software designs, code, and documentation. It is imperative that we make good decisions early in the lifecycle, and uncover problems in the requirements and architecture as the architecture and engineering artifacts are developed.

Predictable System-of-Systems Behavior

Given that interconnected battle-management solutions in the system-of-systems environment are separately designed and developed on various operating platforms in different languages, predicting battle-management behavior of the system-of-systems is not possible. As a rule, battle management is still executed at the system level rather than the desired system-of-systems level. Another factor that contributes to the challenge involved in predicting battle-management behavior is the acquisition practices currently employed in DoD. The increased pressure to rapidly move product into the operational battlespace tends to channel program managers into focusing on achieving functionality as quickly as possible. As such, the development community responds with a hurried and oftentimes inadequate design phase and follows with an intense period of coding. In the rush to rapidly develop a product, one can fall into the trap of exclusively seeking some level of achieved capability while ignoring the behavior of the software.

System-of-systems functional and performance expectations of the users continue to increase as the acquisition community continues to develop and field the products of C4 systems and weapon-systems integration. The class of systems in which battle-management systems are contained is called Command, Control, Communications, and Computers (C4) systems. Typically, C4 systems are non-real-time systems. Traditionally, weapon systems are real-time systems. If we are to match the performance of the weapon systems and avoid the negative impact of forcing synchronization of the battle manager with the weapon system for messaging, then we must develop the battle manager as real-time software.

Another factor that contributes to the challenge involved in predicting battle-management behavior is the acquisition practices currently employed in DoD. The increased pressure to rapidly move product into the operational battlespace tends to channel program managers into focusing on achieving functionality as quickly as possible. As such, the development community responds with a hurried and oftentimes inadequate design phase and follows with an intense period of coding. In the rush to rapidly develop a product, one can fall into the trap of exclusively seeking some level of achieved capability while ignoring the behavior of the software.

The battle-manager behavior must be predictable and must respond immediately to hostile actions of potential adversaries. The warfighters must have confidence that the battle-management software will perform its critical tasks as designed, and will not exhibit inappropriate system behavior such as reporting false ballistic-missile threats and issuing engagement commands for non-existent ballistic-missile threats.

In the operational battlespace, the battle-management software will control the behavior of various weapon systems over a global control network. Based on processed information within the battle manager, it will report ballistic-missile threats to all layers of management up to the President of the United States. The battle manager will assign a weapon to engage each detected ballistic-missile threat and order the engagement of each ballistic-missile threat.

It is neither feasible nor cost-effective to rewrite all the software in the sensors, weapons, and command and control components. As previously discussed, we typically develop each system independently of all the other systems. As such, each system's software is different with respect to architecture, design, missions and functions, language, operating systems, and persistent data storage schemes.

We believe that what is possible is to develop the battle manager for a missile defense system-of-systems with predictable system behavior. Given that the battle manager determines the existence of a ballistic-missile threat and orders the engagement of the ballistic-missile threat, we believe that predictable system behavior of the battle manager will significantly improve the overall predictability of the missile defense system-of-systems.

As the initial step to the battle-manager development, we recommend performing a domain analysis of the battle-management functions. During this type of analysis, software engineers could derive warfighter usage requirements from battle-management use cases. They could refine the use cases as we develop sequence diagrams to depict the messaging requirements among the derived classes from the use cases. Software engineers could develop a state diagram for the battle manager to identify the desired battle-management behavior. To conclude the domain analysis, they should identify and verify assumptions on battle-management operations to support the development of functional specifications.

We recommend that software engineers develop a sufficient amount of information to automatically produce test cases for the implementation. Otherwise, they run the risk of developing so-called "cartoon models" that are only useful for drafting and refining potential solutions. Software engineers need to develop test-ready models of the battle manager. In order to be testable, a model should contain all the features of the battle manager, preserve sufficient detail that is critical for discovering faults, and faithfully represents the essential states, actions, and transitions in the state diagram.

From the iterative review and refinement of these artifacts, software engineers could develop detailed specifications that focus on defining battle-management behavior and achieving battle-management goals. To achieve the level of desired predictable battle-management behavior, software engineers could develop a formal representation that captures the desired system behavior of the battle manager and verify the formal representation against the expected properties. They should consider the use of logic to describe the battle-manager's functional specifications.

We recommend the verification of the functional specifications with the use of a model-checking tool to determine the degree of system behavior predictability with respect to state transitions and tolerance to battlespace environmental variables. Model checking is not a proof of correctness; however, model checking involves creating functional models of a system and analyzing the model against the formal representations of the desired behavior. The verification should focus on ensuring that the battle manager can meet the specifications and exhibits the desired behavior. (N.B.: For this paper, we will define model checking as the systematic approach for testing functional assertions and substantiating the desired system behavior in the model.)

Software engineers might design test oracles that contain the full range of battle-management variables that are both within and outside the expected range of operational values for the ballistic missile defense. The automated tool will determine whether the logic is appropriate to reach each state and whether the logic prevents reachable states inappropriately. The result of the state-based model verification will be the substantiation or repudiation of the desired battle-manager behavior.

Software engineers should incorporate assertions and error-handling schemes developed from the functional model and verified by the model-checking effort into the battle-manager software. The error-handling schemes for breaks in logic will benefit the warfighters by either developing an automated logic-break recovery or notifying the warfighters of required manual actions.

Conclusions

It is our belief that software engineers can develop a dependable system-of-systems that fulfills warfighter requirements. We believe that acquisition agencies within DoD can develop system-of-system software architectures in the acquisition of such systems with the potential benefits of acquiring systems on time, within budget, and with the desired level of capability as defined by the warfighters. We believe that software engineers can develop a conceptual software architecture that describes a logical organization of proposed software modules.

We believe that software engineers can develop various artifacts that help software engineers understand the system-of-systems behavior rather than depending on voluminous system requirements specifications of the written word. We believe that the combination of employing formal methods in the requirements phase and model checking the functional model formed by the formal methods will result in a significant increase of dependable software as compared to our current “sticks and circles” interconnectivity techniques.

If we choose to employ the techniques of software architectures and formal methods in our system-of-systems development, then we can hope to improve on the dismal record of our software developments. More importantly, we can hope to provide more useful software products to our customers while reducing the time and cost to develop our products. As in all human endeavors of any consequence, keen insight, careful reasoning, and deliberate planning are the keys to a successful outcome. In the absence of these activities, we are surely doomed to failure.

The Growing Need for Trustworthy Model and Simulation Software

Kevin J. Greaney
Missile Defense Agency
kevin.greaney@mda.osd.mil

Software[9] enables the political [1], economic [2], psychological [3], [4], and military [5] centers of United States national power. The United States military center of national power, manifested in the Department of Defense[10] (DoD), employs software-intensive systems[11] in a wide variety of missions, tasks, and functions, including models and simulations[12] (M&S), supporting the Department's national defense mission. The Department requires credible[13] M&S supporting confidence[14] in the simulation results provided to Department- and National-level decision-makers, especially in high-risk, mission-critical decisions [6]. Establishing credibility in Department simulations involves many disciplines and knowledge areas including software engineering[15], processes[16], quality[17], product management[18] and architecture[19].

National- and Department-level decision-makers expect credible Department of Defense M&S to provide them confidence in the simulation results, especially for mission-critical and high-risk decisions supporting National Security. The Department developed many of these large-scale, software-intensive simulation systems autonomously over time, subjected to varying degrees of funding, maintenance, and life-cycle management practices. The Department's complex organizational dynamics, and complicated acquisition procedures [7, 8, 9, and 10] also impact the level of M&S credibility, at times adversely.

There is a new sense of urgency in the Department for credible M&S, and an increased emphasis on improved confidence in simulation results [11]. In the past, models and simulations played secondary and tertiary roles supporting the Department's decision-making process. Today, models and simulations are Department corporate assets [6], evolving into primary enablers directly supporting an information age national security strategy [1]. The Department's support for this strategy and supporting objectives requires confidence, credibility, security and interoperability in M&S, which are "broadly applicable to the full range and scope of missions and activities across the breadth of DoD operations" [12] and enable the Department's execution of critical missions, tasks, and functions [13].

---

[9] Software includes the computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system [24].

[10] Department of Defense will be referred to by the terms Department or DoD, used interchangeably throughout this document.

[11] A software-intensive system is any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole [25].

[12] Models and simulations (M&S) – In this research models and simulations (M&S) refers to mathematical abstractions and software implementations, as opposed to the term modeling and simulations used to identify an analytical problem solving approach [26].

[13] Credible means 1. Capable of being believed; believable; plausible; 2. Worthy of confidence; reliable. Plausible is defined as 1. Seemingly or apparently valid, likely, or acceptable. Reliable is further defined as 1. That can be relied upon; dependable [27].

[14] *Confidence* is defined as 1.Trust or faith. 2. A trusting relationship. 3. Something confided. 4. A feeling of assurance, especially of self-assurance. 5. The assurance that someone keeps a secret [27].

[15] Software engineering is the application of a systematic, disciplined, quantifiable approach to the development; operation, and maintenance of software; that is, the application of engineering to software [24].

[16] Process is a sequence of steps performed for a given purpose: for example, the software, development process [24].

[17] Quality is 1). The degree to which a system, component, or process meets specified requirements; 2). The degree to which a system, component, or process meets customer or user needs and expectations [24].

[18] Product management includes the definition, coordination, and control of the product characteristics during its development cycle [24].

[19] Architecture means the organizational structure of a system or component [24].

Mission-critical M&S software[20] uses supporting national security policy decisions include the emerging Homeland Defense security policy [3, 4, 14, 15, 16, and 17] and complex, high-risk, defensive systems such as ballistic missile defense [18], designed to counter asymmetric twenty-first century threats against the American Homeland, including global terrorism and weapons of mass destruction. President Bush recently signed the *National Strategy For Homeland Security* [4] emphasizing six mission-critical areas supporting the Nation's defense including the protection of critical infrastructure and key assets. Within the critical infrastructure mission area, the President identified eight major initiatives including the development of a national infrastructure protection plan, securing cyberspace, and harnessing "the best analytic and modeling tools to develop effective protective solutions" [4].

From a strategic perspective, "Modeling and simulation are simply a means to an end" [18]. Confidence in the high-risk, mission-critical results produced from the Department's simulations employed as a means for National Security necessitate the highest levels of credibility. The Department's draft *Modeling and Simulation Strategic Plan* [11] reflects this new emphasis on the warfighter with the first Department M&S mission statement: "Exploit modeling and simulation to transform the warfighting capabilities of the United States" [11]. However, systemic problems with distributed interoperability of these non-trivial simulations in federations' continue, and current techniques, procedures, and tools have not achieved the desired results. Recent Defense Model and Simulation Office study initiatives [19, 20, 21, and 22] indicate many of the persistent systemic problems remain, and identify the true scope of the issues, challenges, and difficulties faced by the Department's M&S community to develop credible, trustworthy M&S software and instill confidence in simulation results.

References

[1] Bush, G., *National Security Strategy of the United States of America*, President of the United States, The White House, Washington, DC, September 20, 2002.

[2] Research Triangle Institute, *The Economic Impacts of Inadequate Infrastructure for Software Testing Final Report*, RTI Health, Social, and Economics Research, Research Triangle Park, NC, May 2002.

[3] Bush, G., *The Department of Homeland Security*, President of the United States, The White House, Washington, DC, June 2002.

[4] Bush, G., *National Strategy for Homeland Security*, President of the United States, The White House, Washington, DC, July 2002.

[5] Secretary of Defense, *Quadrennial Defense Review Report*, Department of Defense, Washington, DC, September 30, 2001.

[6] Department of Defense, Under Secretary of Defense for Acquisition and Technology, *DoD Modeling and Simulation Management*, DoD Directive 5000.59, Under Secretary of Defense for Acquisition and Technology Washington, DC, January 4, 1994, with Change 1, January 20, 1998.

[7] Birkler, J., Smith, G., Kent, G., *An Acquisition Strategy, Process, and Organization for Innovative Systems*, RAND, Santa Monica, CA, 2000.

[8] Swank, W., Alfieri, P., Giley, C., *Acquisition Trend Metrics in the Department of Defense*, TR 1-00, Defense Acquisition University Press, Fort Belvoir, VA, October 2000.

[9] Wolfowitz, P., *Cancellation of DoD 5000 Defense Acquisition Policy Documents*, Deputy Secretary of Defense memorandum, Washington, DC, October 30, 2002.

[10] Wolfowitz, P., *Defense Acquisition*, Deputy Secretary of Defense memorandum, Washington, DC, October 30, 2002.

[20] Critical software is software whose failure could have on safety, or could cause large financial or social loss [24].

[11] Department of Defense, *Department of Defense Modeling and Simulation Strategic Plan 2003-2012, (DRAFT),* Office of the Undersecretary of Defense For Acquisition, Technology, and Logistics, Washington, DC, April 2003.
[12] Department of Defense, *DoD Modeling and Simulation (M&S) Master Plan (DRAFT),* Department of Defense Instruction No. 5000.71, Office of the Director of Defense Research and Engineering, Defense Modeling and Simulation Office, Washington, DC, May 24, 2001.
[13] Shalikashvili, J. *Shape, Respond, Prepare Now: A National Military Strategy for a New Era,* The Joint Chiefs of Staff, Washington, DC, 1997.
[14] Fields, C., Odeen, P., Poste, G., *Protecting the Homeland: Report of the Defense Science Board 2000 Summer Study, Executive Summary,* Volume I, Office of the Undersecretary of Defense For Acquisition, Technology, and Logistics, Washington, DC, Feb. 2001.
[15] Brennan, R., *Protecting the Homeland, Insights From Army Wargames,* RAND Arroyo Center, Santa Monica, CA, 2002.
[16] Bush, G., *Securing the Homeland Strengthening the Nation,* President of the United States, The White House, Washington, DC, 2002.
[17] Clarke, R., Schmidt, H., *A National Strategy to Secure Cyberspace,* President's Critical Infrastructure Board, The White House, Washington, DC, September 18, 2002.
[18] Bush, G., *President Announces Progress on Missile Defense Capabilities,* President of the United States, The White House, Washington, DC, December 12, 2002, accessed on Dec. 23, 2002 from http://www.whitehouse.gov/news/releases/2002/12/20021217.html.
[19] Department of Defense, *C4ISR Information Superiority Modeling and Simulation Master Plan,* Office of the Assistant Secretary of Defense Command, Control, Communications, and Intelligence, Washington, DC, March 12, 2002.
[20] Hartman, F., Collier, S., Gardner, D. *Integration Task Force Assessment,* Defense Modeling and Simulation Office, Department of Defense, Office of the Director of Defense Research and Engineering, Washington, DC, July 10, 2001.
[21] Hartman, F., Collier, S., Gardner, D. *Integration Task Force Report,* Defense Modeling and Simulation Office, Department of Defense, Office of the Director of Defense Research and Engineering, Washington, DC, August 20, 2001.
[22] Moss, T., Wrigley, J., Dickinson, R., *Warfighter M&S Needs Assessment of the Unified Commands and Selected Supporting Commands (Final Report*), Modeling and Simulation Information and Analysis Center, Defense Modeling and Simulation Office (DMSO) Washington, DC, November 17, 2000.
[23] Moss, T., *DoD M&S Master Plan Baseline Assessment,* Modeling and Simulation Center Information Analysis Center presentation, Defense Modeling and Simulation Office (DMSO), November 6, 2000.
[24] IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology,* Institute of Electrical and Electronics Engineers, New York, NY, 1990.
[25] IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems,* Institute of Electrical and Electronics Engineers, New York, NY, 2000.
[26] Glasgow, P, Muessig, P., Sikora, J., *Department of Defense Verification, Validation and Accreditation (VV&A) Recommended Practices Guide,* Office of the Director of Defense Research and Engineering Defense Modeling and Simulation Office, Washington, DC, November 1996.
[27] Webster, *Riverside Webster's II New College Dictionary,* Houghton Mifflin Company, Boston, MA, 1995.

Trustworthy Software

Charles C. Howell
The MITRE Corporation
howell@mitre.org

From the Workshop call for participation:

> [..] achieving high levels of trustworthiness in software has been an elusive goal, as has defining precisely what is meant by the term "trustworthy."

I certainly agree enthusiastically with this description, and with the sense of urgency communicated by the workshop organizers. I appreciate the opportunity to participate in the workshop.

The request for a position paper states:

> If you would like to participate in this workshop, we are requesting that all attendees draft a one-two page position paper that presents their views on two issues: (1) what does "trustworthy software" mean to you? and (2) how can a national software agenda be created to ensure that commercial software is of higher integrity?

It's a slippery slope down to nit picking and over-precision to quibble about terms, I admit, but even in the workshop description several terms are used seemingly to mean similar things: - trustworthy, reliable, and high integrity. Microsoft uses the term "trustworthy computing," and of course there are reasonably well-understood (or at least agreed upon) terms such as dependable, high-confidence and high-assurance software.

My suggestion for the workshop is to settle on one widely available term and definition for the desired quality or goal we intuitively understand when we encounter the phrase "trustworthy software." Admittedly, some definitions are not of much help, for instance, “dependability” defined by IFIP WG-10.4 as "The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers."

The aspect of trustworthy software I think deserves the focus of attention at the workshop is the implied quality of "few surprises" - the user (end user, or developer integrating a component into a system, etc.) has good reason to expect that he or she understands the behavior of the software under all anticipated ranges of inputs and environments. This raises questions about how well the operational context and likely evolution of the system are understood. It also implies that in order to be trustworthy, software probably has to be robust in the face of unexpected uses and evolutionary change, or at the least if it is fragile it fails in understood and not surprising ways.

For example, an air traffic control system that occasionally "crashes" - just stops working, the classic blue screen of death - is clearly less trustworthy than one that is highly available and reliable. But, I suspect an ATC that occasionally displays hazardous and misleading information - that doesn't crash, just shows plausible but wrong information to the controllers - is much less trustworthy than the one that crashes. The surprise is not that it crashed but that it was wrong and we didn't know it, much more damaging to trust.

If trustworthy software is "low surprise" software, the focus has to turn to design for transparency, for better characterization of robustness and of the expected operational environment, and a better understanding of the sources of surprise in current software. This leads me to the second set of issues in the call for participation: elements of a national software agenda. I am heavily influenced here by previous work that clearly relates to this workshop. Three previous studies/workshops of relevance IMHO:

- "Trust in Cyberspace," Committee on Information Systems Trustworthiness, National Research Council (1999), http://www.nap.edu/readingroom/books/trust/
- "Workshop on New Visions for Software Design and Productivity: Research and Applications," Software Design and Productivity Coordinating Group, http://www.isis.vanderbilt.edu/SDP/
- NSF Workshop on a Software Research Program for the 21st Century, Greenbelt, Md. 15-16 Oct. 1998, http://www.cs.umd.edu/projects/SoftEng/tame/nsfw98/

In particular, a succinct summary of this point is from the NSF Workshop:

> Finding 1: Current software has too many surprises. The sources of surprise are poorly understood.
> Recommendation 1: Emphasize empirical research aimed at understanding the sources of software surprises.

I think it is obvious that we cannot really improve what we do not understand. I would argue that we do not understand many of the real technical sources of surprise, especially as we integrate larger and more complex distributed networked software based on COTS and legacy components we don't have full access to.

An agenda that jumps to technical fixes *du jour* or to classic management proscriptions will not tackle our engineering ignorance. An agenda that focuses on empirical study, on dynamic and static analysis of various software artifacts, on better visibility into operational history of real software, on models of software properties that can be validated and that increase our ability to predict behavior of complex systems before they are built…these are aspects of what I would encourage the workshop to recommend for a national software agenda.

A final "cite bite" that I think is as accurate today as when written[*]:

> Software engineering is characterized by many strongly held convictions as to what techniques produce the most reliable systems. Almost without exception, these convictions are anecdotal, having evolved in a near complete absence of any supporting data, but they are considered intuitively self-evident.
>
> In recent years, there has been reliable data published on software systems. Analysis of this data against a number of these widely held beliefs suggests that they are either in considerable doubt or simply not true. This has many implications for the engineering of more reliable software systems.

[*] Les Hatton, "Unexpected (and Sometimes Unpleasant) Lessons from Data in Real Software Systems," in *Safety and Reliability of Software Systems*, 12th Annual CSR Workshop, 1995.

Trustworthy Software

Ronald J. Kohl
R. J. Kohl & Associates, Inc
rjkohl@prodigy.net

1. What does "trustworthy software" mean to you?

I would offer the following definition of 'Trustworthy Software': software for which there is a very high degree of confidence that this software will do exactly what it is required to do or what it is advertised to do. No more and no less.

Note that this definition introduces some challenging terms. First is 'high degree of confidence'. I do not offer a measure by which such a criteria could be attained and I would hope that this workshop would contribute to addressing the concept of 'high degree of confidence'. Next is 'no more, no less'. This is a significant challenge in COTS s/w in that there may be vendor-provided s/w capabilities in a given product that not required/desired by the product acquirer/integrator.

2. How can a national software agenda be created to ensure that commercial software is of higher integrity?

In this area, it may be helpful to revisit the COTS hardware industry (e.g., chips and boards), in that many years ago, some of these same challenges and issues existed with COTS hardware. But that industry took certain steps (standards for functionality, standards for interfaces, standards for packaging, etc) that significantly increased the quality and integrity of those COTS hardware products. There may be some lessons to be learned from the experiences of the COTS hardware industry that could contribute to a national approach towards improving the integrity of COTS software.

Regarding what could be proposed in the near-term to address these issues, it may be possible to establish a national 'COTS software-lessons-to-be-learned' database (or establish pointers to existing such DBs) so that both vendors and users could contribute information about what went well and what did not go well for a given COTS software product in a given application. It should be noted that some products already have robust 'user groups' that include 'user forums' and even some such repositories of information about a product. It is not unusual for the COTS product vendor to participate in and contribute to such 'user groups.' This may be a model to be emulated across the COTS s/w industry.

Additionally, the concept of defining some 'standard test suite' for a given product is worth considering. Such a 'standard test suite' would not, of course, be able to test all possible combinations of COTS s/w product installed on computer h/w, possibly integrated with other s/w products, all to achieve some domain specific purpose, but some subset of these various combinations could be considered. Such an approach would require involvement by both the vendors and the users, as well as integrators and probably other stakeholders.

Can Aspects of Software Trustworthiness Be Made Computable?

Richard C. Linger
Software Engineering Institute
rlinger@sei.cmu.edu

Present Capabilities

Many aspects of trustworthiness depend on the correctness of software with respect to its specifications. It has been economically and technically possible for some time to develop small systems, or subsystems of large ones, at a fielded defect rate of under 0.1 errors/KLOC. For example, Cleanroom software engineering [1] combines function-based design and verification with usage-based testing to provide statistical certification of software fitness for use. These foundations are being extended to network-centric systems [2] and automated analysis of software functionality [3]. Other technologies, such as proof checking and model checking, provide computational support for approaching zero defects and validating trustworthiness properties. However, such methods are not widely used in industry.

Possible Future Capabilities

A principal characteristic of the evolution of engineering disciplines is replacement of empirical methods with computational methods. This evolution tracks improvements in understanding of the fundamental properties of engineering subject matter. A century ago, the Wright brothers experimented with wind tunnels to evaluate airfoils, a task carried out today by computational fluid dynamics. Computational methods can give confidence when stakes are high and engineering questions must be objectively decided. This observation motivates the question of whether progress can be made in making aspects of software trustworthiness computationally decidable.

Modern engineering disciplines employ rigorous methods to evaluate the expressions that represent and manipulate their subject matter. For example, the equations that represent orbital motion or electromagnetic fields allow engineers to understand and predict the behavior of their designs. However, software engineering is an exception to this rule; it has no practical means to fully evaluate the expressions it produces. In this case, the expressions are programs, and evaluation means understanding their full behavior, whether right or wrong, intended or malicious.

In today's state of practice, it is difficult for a software engineer can say for sure what a sizable program does in all circumstances of use without an impractical expenditure of time and effort. Business, government, and defense must depend on large-scale systems composed of programs whose full behavior and security exposures are not reliably known. This inability to fully understand the behavior of software lies at the heart of many seemingly intractable problems in system development and operation.

It is difficult to evaluate propositions of software trustworthiness when full behavior is not known. Testing is an empirical process that provides at best statistical, and at worst anecdotal, evidence of trustworthiness. Formal methods such as theorem proving and model checking can be employed to achieve higher levels of assurance, but are difficult to scale up in scope and coverage. Questions of trustworthiness address properties of functionality, security, safety, fault tolerance, etc., whose answers involve understanding how programs behave. In the absence of this knowledge, other factors, such as organizational capabilities, development processes, and personnel qualifications, are often evaluated to help assess trustworthiness.

Programming today is a largely empirical activity characterized by much experimentation and discovery in trying to get programs to behave as envisioned. Code is the primary form of program expression, even though it is the semantics of program behavior that matters to users in the end.

Imagine for a moment that new technology could be developed and embodied in automated tools to extract the net functional behavior of programs through computational means [3, 4]. That is, the behavioral semantics of programs expressed in common languages could be computed to arrive at procedure-free expressions of how inputs are transformed into outputs in all circumstances of use. These as-built specifications would be available in behavior repositories for analysis by human and automated means for aspects of functional correctness, security, safety, and other properties of trustworthiness. Even small-scale behavior computation could change many activities in software engineering. Programmers could compute the behavior of programs under development to assess evolving functionality. Acquisition organizations could inspect behavior repositories for conformance to requirements and specifications. Programs could be found to be unacceptably incorrect, or even acceptably insecure given a benign operating environment. Behavior would be known, and informed engineering decisions could be made.

Many formidable barriers stand in the way of such a capability for computing software behavior. For example, no general theory for abstracting the behavior of loops can exist. Here, pattern recognition may provide an engineering solution. The sheer size of programs may overwhelm the behavior computation process. Here, a view of programs as rules for functions or relations may permit a stepwise behavior abstraction process that preserves net effects while leaving local processing details behind. The complexity of programs may produce behavior expressions that are difficult to understand. Here, recognition that the language for expressing behavior to users need not be executable can permit a full range of abstraction techniques for human understanding. Overcoming these barriers could help address some risks to trustworthiness, but certainly not all. Hardware failures, operational mistakes, and malicious intervention will always require risk management, and purely technical solutions will never be sufficient.

In any event, this imagined capability illustrates a possible evolution of software engineering toward computational semantics embedded in a new generation of intelligent tools that could help decide some aspects of trustworthiness in a definitive way. Can such tools be built? That is an open question. An effort is underway to develop a behavior computation tool for analyzing the functionality of malicious code as a first small step toward answering this question.

---

1. S. Prowell, C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Process,* Addison-Wesley, Reading, MA, 1999.

2. R. Linger, M. Pleszkoch, G. Walton, and A. Hevner, *Flow-Service-Quality (FSQ) Engineering: Foundations for Network-Centric System Analysis and Development,* Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2002-TN-019.

3. R. Linger and M. Pleszkoch, **"**Function Extraction (FX) Technology: Automated Calculation of Program Behavior for High-Assurance Systems," *Proceedings of High Assurance Systems Engineering conference (HASE 2004),* Tampa, FL, March, 2004, IEEE Computer Society Press.

4. M. Pleszkoch and R. Linger, "Improving Network System Security with Function Extraction Technology for Automated Computation of Program Behavior," *Proceedings of Hawaii International Conference on System Sciences-38 (HICSS-38),* Hawaii, January, 2004, IEEE Computer Society Press.

Gaining the Trust of Stakeholders in Systems-of-Systems: A Brief Look at the Ballistic Missile Defense System*

J. Bret Michael
Naval Postgraduate School
bmichael@nps.edu

1. Introduction

A *system-of-systems* is a federation of legacy and developing systems that provide an enhanced capability greater than that of any of the individual systems within the system-of-systems.[1] One would hope that a system-of-systems will be long-lived, as past experience has shown that a significant investment must be made to develop such systems. Irrespective of their longevity, these systems can exist in a continual state of flux: they must adapt as the environments within which they operate change, that is, the things in the environment may be neither stable nor closed.

The U.S. Department of Defense (DoD) is becoming increasing reliant on systems-of-systems. An example of a system-of-systems is the Ballistic Missile Defense System (BMDS), which will serve as a global shield against threat ballistic missiles. The system will be composed of legacy and new systems provided by both the U.S. and its allies: most of these legacy systems were not designed to be readily interoperable with other systems.[2] The BMDS will provide enhanced capabilities over those that can be provided separately by the individual systems from which the BMDS is composed. For instance, the BMDS will provide for conducting network-centric warfare, such as forward cueing of sensors and engaging threat objects on remote, both of which are examples of cooperative-engagement capabilities (CEC). Billions of U.S. dollars have been spent already on requirements analysis, prototyping, and test and evaluation, and approximately $10 billion has been requested of Congress by the Missile Defense Agency for the coming fiscal year.

The architecture of the BMDS must permit the insertion and extraction of components; examples of components are sensor, weapon, command-and-control (C2), and battle-management (BM) systems. For example, in contrast to its medium-range *Taepo-dong*-1 missile, North Korea's newly developed *Taepo-dong*-2 is believed to be an intercontinental-class multistage missile capable of delivering a nuclear weapon. The BMDS sea-, air-, land-, and space-based systems will need to be configured to counter the threat posed to the U.S. and its allies by the *Taepo-dong*-2, in addition to future ballistic-missile threats. Thus, it is evident that the architecture of the BMDS

* This research is supported by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

[1] It is the value-added aspect of systems-of-systems that distinguish them from other types of systems. When developing a system-of-systems, it is desirable to explicitly articulate the integration requirements rather than emergent properties and then ensure that these requirements are provided by the system-of-systems. An *emergent property* is a characteristic quality of a system-of-systems that can only arise through interaction among the systems comprising the system-of-system. An *integration requirement* is an emergent property that the system should provide.

[2] The U.S. Navy's Aegis Combat System, for instance, was architected with a tight coupling among its weapon, battle-management, command-and-control, and sensor subsystems, making it difficult to provide for interoperation between Aegis and other systems without resorting to the kludge of potentially brittle automated translators, or re-architecting the entire Aegis system; here we use the term "kludge" to have the meaning ascribed to it by Jackson Granholme: "An ill-assorted collection of poorly matching parts, forming a distressing whole" ("How to design a kludge," *Datamation*, Feb. 1962, pp. 30-31).

must outlive its component systems, providing for plug-and-play of systems. It is not practical from a political, economic, or national-security perspective to constantly re-architect the BMDS.[3]

2. Trustworthiness vs. Dependability

A system-of-systems such as the BMDS must be dependable. A *dependable system-of-systems* provides—under real-world operational profiles—the value-added services and guaranteed properties for each of its modes of operation for the current or a target configuration of its component systems. Value-added services and system properties are specified as system requirements, which are in turn refined into lower level system artifacts (e.g., architectures, designs, algorithms, and executable assertions). In the context of ballistic missile defense, examples of services are discrimination and creation of tracks. Dependability properties characterize a system-of-systems' behavior, such the level of integrity with which input data is handled (e.g., checking that raw sensor data is not corrupted during local discrimination), the degree of safety with which hazards are dealt (e.g., mitigating hazards that could lead to an inadvertent missile launch), the tolerance of the system to faults (e.g., recovering from arbitrary failures exhibited by a battle manager), and the availability of the system's services when they are requested (e.g., having available the minimum number of battle managers for engaging threat missiles during the heat of battle).

A system-of-systems can be dependable but not trustworthy, and vice versa, but the BMDS, like many other systems used for safety-critical or national-security purposes, must be both dependable and trustworthy. A *trustworthy system-of-systems* is one in which a stakeholder[4] places trust in the credibility and sufficiency of the claims and supporting arguments that the system is dependable; the claims and their supporting arguments are collectively known as a "dependability case." If the stakeholder judges the dependability case to be either unconvincing or inadequate, then that person or group will likely place a low degree of trust in the system-of-systems to dependably deliver its services and guaranteed properties. Concerns about the adequacy of the services (i.e., functionality) and properties of the BMDS have been raised recently by members of Congress[5] and the U.S. Government Account Office (GAO): their trust in the system is low due to perceived—and in some cases actual—deficiencies in the test and evaluation of the initial defensive capability to be deployed as part of Initial Defensive Operations (IDO) in 2004.[6] One of the key criticisms in the recent GAO report is that capabilities and properties of the initial system have not been tested under a sufficient and realistic set of engagement scenarios. However, it is not economically or technically feasible using today's state-of-the-art techniques to perform the extensive operational testing recommended in the GAO report. Even if it was feasible to do this for the IDO configuration, testing will have to be repeated for future configurations of the system.

---

[3] This is a high-visibility, multi-billion dollar system that will act as a deterrent to missile attacks. It would be politically unpalatable to spend ten's of billions of U.S. dollars to significantly re-architect the system or start from scratch. Rather, the system needs to be dynamically reconfigurable so that it can readily defend against new threats, incorporate new technologies and systems, and accommodate changes in political and military doctrine, strategy, and tactics. In addition to "plugging in" new systems, systems that no longer serve to provide a value-added capability will be "unplugged" from the BMDS.

[4] A *stakeholder*, in this context, is a person or group that has an interest in the dependable behavior of a system-of-systems. There are many stakeholders in BMDS, such as the director of Operational Test and Evaluation for the U.S. Department of Defense, the director of the Missile Defense Agency, the head of the U.S. Strategic Command, members of Congress, allied nations, the system developers (i.e., contractors), and the general public, to name a few.

[5] Missile Defense Still Uncertain; Test Chief Unsure System Would Work on N. Korean Target, *Washington Post*, 12 Mar. 2004, p. A6.

[6] U.S. General Accounting Office. Missile Defense: Actions are Needed to Enhance Testing and Accountability. Report GAO-04-409, Washington, D.C.: U.S. Government Printing Office, Apr. 2004.

3. Unanswered Questions

Software already plays the leading role in providing the ability of systems-of-systems to adapt to new or modified requirements. The current version of the BMDS relies on the "lashing" of systems together via the use of software-based "translator boxes." The translators, however, are brittle: they must be modified each time a change is made to any of the component systems.

Researchers at the Naval Postgraduate School are helping the Missile Defense Agency develop an architecture that can outlive the system's components, without relying on the use of translator boxes. The architecture will serve as a firmly engineered foundation upon which to build dependability and trustworthiness into the BMDS. We advocate a departure from "business as usual," by requiring Spartan and Draconian designs of systems-of-systems. The general idea is to first distinguish which system requirements are stable from those that are expected to change, and then to institutionalize the invariant part of the principles of operation of the system in a Spartan safety kernel.[7] The next step is to enforce on system developers a Draconian global structure which provides safety with non-interference guarantees, with the aim of making the visible dependencies much less than potential dependencies: fault containment will be handled at boundaries and invisible interactions among components will not be permitted.

There is a growing body of experience to guide us in architecting, designing, and assessing systems-of-systems. However, there is little empirical evidence upon which to judge the efficiency or effectiveness of proposed principles, mechanisms, or practices. In addition, we need to answer fundamental questions such as the following:

- Given that system-of-systems will need to adapt to new or modified requirements, is it possible to obtain a sufficient set of system requirements from which to define the dependability properties for the system?
- How will we measure dependability properties? System safety, for instance, relies on predictability: there is a need to know what the system must guard against (i.e., hazards). Given a lack of environment predictability, how does one handle unanticipated hazards?
- How will we perform assurance activities against systems-of-systems that are moving targets? Assurance work is typically performed against a fixed known model, not a radically changing environment. Adaptive systems can have lots of configurations and are therefore hard to characterize: each instance of a component has a different view of the system. However, is it possible to create a sufficiently large closed world so that one can deal with all of the system hazards, or validate an upper bound on the probability of a system failure leading to a given hazard?
- How will we organize, maintain, and present dependability cases to different types of stakeholders of complex, large systems-of-systems?

In addition to these questions about the technical aspects of developing systems-of-systems, we need to think about non-technical aspects too, such as how to change defense system acquisition policy to encourage a system-of-systems philosophy in the systems engineering community, rather than continued development of monolithic throwaway systems, or how to discourage the starting-from-a-clean-slate (a.k.a., "green-field development") mentality.

[7] The Spartan safety kernel provides liveness properties with service guarantees for those services needed to achieve critical requirements within the strict time budgets for intercepting threat ballistic missiles. In the case of BMDS, we found these stable requirements to be those associated with the battle manager.

Modest, Practical Steps to Encourage Higher Quality Software

Keith W. Miller
University of Illinois at Springfield
miller.keith@uis.edu

What does "trustworthy software" mean?

In 1973, F. Lockwood Morris used the phrase "trustworthy software" while discussing proving compilers correct, attributing the idea to John McCarthy. When people use the phrase recently, they often cite a 1990 paper by Parnas et al.[2]: "We consider a product to be trustworthy if we believe that the probability of it having a potentially catastrophic flaw is acceptably low." That definition seems closely related to others' definition of software safety: "Software is deemed *safe* if it is impossible (or at least highly unlikely) that the software could ever produce an output that would cause a catastrophic event for the system that the software controls."[3] Both definitions use the term catastrophic. A major difference between these definitions and a common definition of reliability is that reliability is defined as the probability of "fault-free operation."[4], and does not distinguish between failures that are catastrophic and failures that aren't.

The term "catastrophe" is not a measurable, technical quantity. The same event might be labeled as a catastrophe for one person and not for others. The loss of a single unpublished manuscript could be a catastrophe for the author, but perhaps not for anyone else. A measure of trustworthiness as defined by Parnas requires a determination of how disastrous is would be for users (and perhaps for different groups of users) if a particular feature fails.

Such difficulties in measuring trustworthiness may lead some is argue that the current software market should decide what software is sufficiently trustworthy. This argument includes the idea that customers will tend to buy software that has what they think has the right combination of quality, price, and features. I disagree with that argument. I think the market for computer software is distorted so that quality software is not as attractive as it might otherwise be. For example, large corporations tend to buy software from large corporations [5]. Furthermore, in software markets, market share is even more important than in other industries [6]. Finally, the current market often does not provide adequate information about software quality so that software consumers can make informed decisions. The next section includes ideas that may help consumers recognize some aspects of software quality.

How can a national software agenda be created to ensure that commercial software is of higher integrity?

I'm on record as favoring a minimum standard for software testing. [7] The idea was lustily attacked and gained little acceptance, and I'll not argue for it further here. I co-authored a paper on third-party evaluations of software quality [8], and I think that idea still has merit. I co-authored a paper that explored the ethical implications of software quality,[9] a paper that at least some people link with the derivation of the term "good enough software." That paper emphasized the importance of good faith negotiations between users and developers about software quality. I think that emphasis is useful in this discussion, but it's difficult to devise a national agenda about good faith. Instead, a more practical strategy may be to increase the information available to software buyers about software quality, in the expectation that higher quality software will do better in the market when consumers can better identify at least some aspects of software quality.

Cem Kaner has written extensively and coherently on the legal environment for software development in the U.S. His "Software Customer Bill of Rights" [10] includes ten clauses, two of which I will quote:

> ***Disclose known defects.*** The software company or service provider must disclose the defects that it knows about to potential customers, in a way that is likely to be understood by a typical member of the market for that product or service.
> ***Mass-market customers may criticize products, publish benchmark study results, and make fair use of a product.*** Some software licenses bar the customer from publishing criticisms of the product, or publishing comparisons of this product with others or using screenshots or product graphics to satirize or disparage the product or the company. Under the Copyright Act, you are allowed to reproduce part of a copyrighted work in order to criticize it, comment on it, teach from it, and so on. Software publishers shouldn't be able to use "license" contracts to bar their mass-market customers from the type of free speech that the Federal laws (including the Copyright Act) have consistently protected.

Both of these suggestions, if enforced, could provide customers with explicit information about the quality of specific software products. The first suggestion may have the unintended side-effect of creating a disincentive for finding defects. However, if the defects discovered after release are included in the public disclosure, that problem is diminished. The second suggestion is timely, since recent legislation seems to have singled out software developers for special protections against criticism, protections that other industries do not have.

A third suggestion for more accurate information about software comes from Valdis Berzins' position paper for this workshop.[11] He contends that now or in the near future we will have automated methods for determining some software characteristics important to quality. He points out that there is little hope for having all such characteristics automatically determined. However, as an incremental step towards improvement, it seems eminently reasonable to exploit existing technologies when possible. Berzins lists twelve possibilities in his paper. Three of the twelve are the absence of memory leaks, the absence of buffer overruns, and the determination that every opened file is closed before the program terminates. Whenever such characteristics can be ascertained in an automated, relatively inexpensive manner, a useful indication of software quality can be determined before software is released.

Berzins suggests that these determinations could be the basis of a certification process. Alternatively, vendors could do the determination themselves, and include their results as part of deliverables (for custom software) and as part of labeling (for mass marketed software). The labeling of software could come to include (either by custom or by regulations) straightforward information about characteristics of quality that have or haven't been attained in the software. This labeling would be similar to labels that already present information about food content, new car gas mileage, and appliance energy consumption.

The suggestions discussed in this paper are modest when compared to the significant quality improvements that might someday be possible. But these modest steps are practical now or in the foreseeable future. My position is that modest, practical steps should be pursued in order to make progress in the near term.

References

[1] F. Lockwood Morris. Advice on structuring compilers and proving them correct. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Boston, Massachusetts (1973), 144 – 152.

[2] David L. Parnas, A. John van Schouwen, Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM* (June, 1990), Volume 33, Issue 6, 636 – 648.
[3] Cigital Labs. Software Safety. http://www.cigitallabs.com/resources/definitions/software_safety.html (accessed March 30, 2004).
[4] Cigital Labs. Software Reliability. *http://www.cigital.com/presentations/zurich98/tsld028.htm* (accessed March 30, 2004).
[5] Martin Brampton. Why Microsoft should be happy about open source. http://www.zdnet.com.au/insight/software/0,39023769,39116181,00.htm (February 18, 2004), (accessed March 30, 2004).
[6] Falk v. Westarp, Oliver Wendt. Diffusion Follows Structure – A Network Model of the Software Market. Proceedings of the 33rd Hawaii International Conference on System Sciences.(2000), http://csdl.computer.org/comp/proceedings/hicss/2000/0493/07/04937039.pdf (accessed March 30, 2004), 1-10.
[7] K. Miller. A modest proposal for software testing. *IEEE Software*, Vol. 18, No. 2 (March/April 2001), 96-98.
[8] K. Miller and J. Voas. An ethical can of worms for software certifiers. *IT Professional*, Vol. 1, No. 5, (Sept./Oct. 1999), 18-20.
[9] W. R. Collins, K. Miller, B. Spielman, and P. Wherry. How good is good enough? An ethical analysis of software construction and use. *Communications of the ACM*, Vol. 37, No. 1 (January 1994), 81-91.
[10] Cem Kaner. Software customer Bill of Rights. Cem Kaner's Blog. http://blackbox.cs.fit.edu/blog/kaner/archives/000124.html (August 27, 2003), (accessed March 30,2004). *Principles of programming languages*. Boston, Massachusetts (1973), 144 – 152.
[11] Valdis Berzins. Trustworthiness as risk abatement. *CNSS Workshop on Trustworthy Software* (April 8-9, 2004), Monterey, CA.

Trustworthy Software

Peter G. Neumann
SRI International
Neumann@csl.sri.com

1. Defining and Measuring Trustworthiness

Here is a simple definition from my DARPA CHATS report in progress, PGN, Principled Assuredly Trustworthy Composable Architectures, 2004 {http://www.csl.sri.com/neumann/chats4.pdf, ps, and html}:

> By *trustworthiness* we mean simply *worthy of being trusted to fulfill whatever critical requirements may be needed* for a particular component, subsystem, system, network, application, mission, enterprise, or other entity. Trustworthiness requirements might typically involve (for example) attributes of security, reliability, performance, and survivability under a wide range of potential adversities. Measures of trustworthiness are meaningful only to the extent that (a) the requirements are sufficiently complete and well defined, and (b) can be accurately evaluated.

Measuring trustworthiness is possible only if there is something to measure. Some folks in the formal methods community believe in explicit formal requirements and formal specifications that can be evaluated for compliance with the requirements. Similarly, software can be evaluated for formal compliance with specifications. We tried this way back in the 1970s.

Metrics are another matter. In measuring reliability, there are many useful attributes (such as mean time to failure of hardware or software) which make sense if the assumptions on which they are based are realistic. However, in measuring security, there are many systemic problems. Mean time to penetration is generally a meaningless concept; posting an attack script on the Web might vitiate any belief in the validity of such a measure. This is particularly a problem with cryptographic algorithms for which brute-force attacks may be totally unrealistic, but where out-of-band exploits can crack the crypto in short order (e.g., keys exposed in memory, or operating system flaws that can result in access to unencrypted text, or differential power analysis that obtains smart-card keys without any internal access to smart card). We know testing is inherently limited.

2. Shipping with Latent Defects

It all depends on what kind of a system you are starting with. Shipping with many known latent defects should not be considered acceptable practice. But that is not good enough. Better architectures and better software engineering are essential. The truth is that rush-to-marketplace generally results in seriously flawed software, and backward compatibility with systems that are inherently flawed complicates matters considerably. At a recent meeting I attended, a Microsoft employee declared that there are no known unpatched flaws in Microsoft releases. (The audience jumped all over him for that.) This is likely to degenerate into religious arguments, e.g., between Microsoft apologists vs. Open-source zealots. (We've been around that bush a few times in recent years.)

3. The State of the Art of Software Engineering?

It is OK in a few pockets of academic research, generally mediocre in many university curricula, and often bleak in practice. (See the Risks Forum {http://www.risks.org} and {http://www.csl.sri.com/neumann/illustrative.html}.)
With only a few exceptions, it tends to ignore security, reliability, integrity, survivability, real-time performance, and large complex systems. Outsourcing high-end software engineering jobs to India and China may make it even worse for the US, although it might in some cases produce better systems! Thus, I conclude that the state of the art is generally inadequate.

4. One UL-like Lab?

Probably not just one. But we need people who understand critical systems, not just a little about security or reliability. So, developing the competency for this is tricky. The Common Criteria might seem like a good starting place, with all sorts of accredited evaluation centers, but the CC itself is not ready for prime time. Also, if the requirements are inadequate against which systems and application are being evaluated, then there are still too many residual problems that can slip through.

5. Conclusions

I conclude that a realistic way to think about obtaining trustworthy systems and networks is to observe what I have described in my CHATS report:
{http://www.csl.sri.com/neumann/chats4.pdf, ps, and html}.
(I'd be very happy to have feedback, as I am about to complete the draft.)
Here is a quote from my DISCEX3 paper, Achieving Principled Assuredly Trustworthy Composable Systems and Networks, providing an early summary of the CHATS report:

> Huge challenges exist with systems and networks that must dependably satisfy stringent requirements for security, reliability, and other attributes of trustworthiness. Drawing on what we have learned over the past decades, our CHATS project seeks to establish a coherent common-sense approach toward trustworthy systems. The approach encompasses comprehensive sets of requirements, inherently sound architectures that can be predictably composed out of well-conceived subsystems, highly principled development techniques, good software engineering disciplines, sound operational practices, and judiciously applied assurance measures. Although such an approach is likely to seem completely old-hat to some researchers and totally impractical to commercial developers, the wisdom thus embodied is seldom used consistently (if at all) in practice; if it were used wisely, much of the untrustworthiness in today's systems would simply disappear.

Security Issues for the National Airspace System

Arthur Pyster and Hal Pierson
Federal Aviation Administration
{arthur.pyster, hal.pierson}@faa.gov

The FAA's operates

> The largest and most complex aviation system in the world, with a safety record second to none.

We must deploy and maintain a complex system-of-systems that is operational, safe, and secure. In order to accomplish this task, we rely on six elements of cyber defense:

- Architectural simplification,
- Element hardening,
- Boundary protection,
- Orderly quarantine,
- Systemic monitoring, and
- Informed recovery.

Although we have make inroads in each of these areas, the increased threats and ever growing technical sophistication demands that we pursue the very latest research in support of our responsibilities.

Architectural simplification. We need more understandable models of security that identify approaches to developing models for running and managing systems in such a manner that the security dependencies between components are immediately understood, even if this means identifying new ways of building software or systems.

There is a need for better techniques and tools for building large-scale secure systems. These techniques and tools must facilitate the construction of hardware and software based information systems as secure systems with availability, integrity, and privacy controlled to a scientifically guaranteed degree against well-funded, intelligent malicious adversaries, even as the systems begin to scale into hundreds of sub-systems and millions of users at various levels of trust.

Element hardening. We need completely known trust levels that can be proven and verified to correctly support both supervisory control of critical systems, and process time-critical information for human decision making at multiple security classification levels, in a manner such that if compromised by any type of malicious cyber attack, we could at least know that the system had failed in a known state.

Boundary protection. We need practical means to eliminate risks from zero day attacks through near immediate detection and pre-emptive prevention of any set of covert information flows or other abnormal behavior.

Orderly quarantine. In addition to simplified architecture, we need tools and techniques to establish pre-planned fallback configurations that provide increased protection in response to attack or perceived threat.

Systemic monitoring. We need better metrics for monitoring and measuring security effectiveness throughout the lifecycle. This requires an appreciation of the economics of information se-

curity technology at least as well as the economics of the general computing software. This understanding, together with metrics, is an essential component of being able to perform quantitative risk assessment in support of security-related decision-making.

Informed recovery. The FAA has successfully dealt with the viruses and worms of the recent past thanks to the design of the systems and the diligence of our system managers in correcting vulnerabilities in a timely manner based on information and guidance provided by our incident response center. However, we recognize that the time in which these actions need to be performed is shrinking; manual intervention inherent in today's procedures has its limits, and automated mechanisms are needed. But before we can trust these automated responses we will need better techniques for understanding and verifying these recovery mechanisms.

Trustworthy Software

Samuel T. Redwine, Jr.
James Madison University
redwinst@jmu.edu

The problems in producing software with the proper confidentiality, integrity, and availability properties compound the existing serious problem of producing quality software. Producing quality software requires personnel with substantial education, training, and experience. The project management involved in a substantial software project is also significant, with management issues causing more failed projects than technical ones. Improving software engineering practices and processes cannot only lead to secure software[8] but to software released with exceptionally few defects, reasonable costs of development, lower maintenance costs, and enhanced reputation for the product.

Security is not just a question of security functionality; the security properties desired must be shown to hold throughout the secure system. Security properties are properties of the entire system. All parts of the system, whether purchased or built in-house, can be relevant.

Producing secure software begins with specifications that define secure behavior exhibiting the required security properties. The specifications must define functionality, and be free of vulnerabilities that can be exploited. The second necessity for secure software production is correct design and implementation to meet the specifications. Software is correct if it exhibits only the behavior defined by its specification – not, as today is often the case, exploitable behavior not specified, or even known to its developers and testers.

Largely driven by needs for high reliability and safety, approaches to building high-dependability software systems exist today. Mating these with security concerns has already resulted in processes used on real projects to produce secure software.

Why do not organizations use known best practices to produce secure software? First, doing what is required to build secure software is difficult at all levels – individual, team, project, and organization. This can be further complicated if out sourcing or acquisition are involved. Second, there is a perception that functionality is more important than quality and security. Finally, there is a belief that developing high-quality, secure software is too expensive. While, for current successful developers, evidence exists that using processes that produce high quality, secure software take less time and effort, upfront costs are associated with changing the way a business operates. This upfront cost, along with the required organizational courage and strain, is one reason few organizations currently use these practices for producing secure software.

While for most organizations, the substantial changes needed to produce secure software will not be easy and the practices positively impacting security are not perfect, for those willing to make the journey substantial results are possible.

[8] "Secure software" means "highly secure software realizing – with justifiably high confidence but not guaranteeing absolutely – a substantial set of explicit security properties and functionality including all those required for its intended usage."

Practices

The problem of producing secure software is both a software engineering problem and a security engineering problem. Overall, the recommendations for producing secure software can be summarized as:

1. Start with an outstanding software engineering process.
2. Augment with sound technical practices relevant to producing secure software.
3. Support with expert management practices.
4. Wherever possible, quantitatively measure the effectiveness of a practice and improve.

First, a very low defect rate software production process is a necessity. Therefore, to start along the path, every organization desiring to produce secure software, whether a software vendor, an organization developing software for internal use, or developing open source software, should **use a process that can predictably produce software with very low specification, design, and implementation defects – less than 0.1 specification, design and implementation defects per thousand lines of new and changed code delivered.** Software producers should then relentlessly improve to further reduce the defect rate. Of the software-development methods reviewed by the National Cyber Security Summit's Security across the Software Development Lifecycle task force Software Process Subgroup, three come reasonably close to meeting the criteria for producing high-quality and secure software. They are, in alphabetical order, the Cleanroom Software Engineering method originally developed by IBM, Correctness-by-Construction by Praxis Critical Systems Limited, and The Team Software Process (TSP) by the Software Engineering Institute.

Cleanroom software engineering includes project management by incremental development, function-based specification and design, functional correctness verification, and statistical testing for certification of software fitness for use [Linger, Mills, Powell]. A number of Cleanroom projects involve classified activities that cannot be reported. Experience shows, however, that fielded defect rates range from under 0.1 defect per 1000 lines of code with full Cleanroom application to 0.4 defects per 1000 line of code with partial Cleanroom application. Many Cleanroom-produced programs have never experienced a first error in testing or in field use.

The Correctness-by-Construction method of Praxis Critical Systems Limited uses mathematically-based approaches for software development and production [Hall]. The method incorporates mathematical models and formal logic to support rigorous software specification, design, coding, and inspection in a process that emphasizes early verification and defect removal. For secure systems, Praxis categorizes system state and operation by security impact and uses architectures that minimize and isolate security-critical functions. The Correctness-by-Construction method has produced near-defect-free software in five projects completed between 1992 and 2003, with delivered defect densities ranging from 0.75 to 0.04 defects per thousand lines of code. Two of the five projects had a special focus on security requirements.

The Software Engineering Institute's Team Software Process (TSP) SM is an operational process for use by software development teams. The TSP methods include comprehensive project management, rigorous design, statistical quality control, formal inspections, and personal defect measurement and management by the software developers. A recent study of twenty projects in thirteen organizations showed that teams using the TSP produced software with an average of 0.06 delivered design and implementation defects per 1,000 lines of code. The average schedule

[SM] Team Software Process and TSP are service marks of Carnegie Mellon University.

error was 6% [Davis]. A recent TSP extension, the Team Software Process for Secure Software Development (TSP-Secure), augments the TSP with defined security practices and special training in security design and quality management methods.

Security must be an integral consideration during product specification and design. Apply formal methods to specification and design of security aspects. Define the security properties of the software. Analyze and review specifications and designs for security. A key element of making this feasible is to design the software so security critical aspects are concentrated to a limited portion of the software. The design should be as simple as possible – possibly sacrificing efficiency – and must be restricted to structures and features that are "safe" and preferably can be analyzed. The design must not assume that the software cannot be broken and should ensure defense in depth or tolerance. This and other security principles should be given close attention.

A programming language with significantly fewer possibilities for mistakes than C or C++ should be used where possible. The programming language should be fully defined, catch all possible exceptions, and have other mistake reducing characteristics such as strong typing.

Static analysis should be used to find known kinds of coding defects. Security testing must be performed including serious attack efforts. Using suitable consideration, software producers should also adopt other practices deemed useful in the subgroup's full report [Redwine]**.**

Acknowledgements

This position paper is based entirely on [Redwine] Copyright © 2004 Noopur Davis, Michael Howard, Watts Humphrey, Gary McGraw, Samuel T. Redwine, Jr., Gerlinde Zibulski, Caroline Graettinger

References

[Davis] Davis, Noopur, and Mullaney, Julia, "The Team Software Process in Practice: A Summary of Recent Results," Technical Report CMU/SEI-2003-TR-014, September 2003.

[Hall] Hall, Anthony, and Roderick Chapman, Correctness by Construction: Developing a Commercial Secure System, *IEEE Software*, January/February 2002, pp.18-25.

[Linger] Linger, Richard, and Stacy Powell, "Developing Secure Software with Cleanroom Software Engineering". Paper prepared for the Cyber Security Summit Task Force Subgroup on Software Process, February 2004.

[Mills] H. Mills and R. Linger, "Cleanroom Software Engineering," *Encyclopedia of Software Engineering, 2nd ed.*, (J. Marciniak, ed.), John Wiley & Sons, N.Y., 2002.

[Powell] Powell, S., C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Process*, Addison Wesley, Reading, Mass., 1999.

[Redwine] Redwine, Samuel T. and Noopur Davis, editors, *Processes to Produce Secure Software: Towards More Secure Software*, National Cyber Security Partnership, March 2004.

Boot-strapping the Industrial State-of-Practice towards Trustworthy Software Development*

Man-Tak Shing
Naval Postgraduate School
shing@nps.edu

1. Introduction

When we type the word "software problem" into any Internet search engine, we can easily come up with dozens of articles reporting the impacts of software problems in our daily lives; e.g., the total disability of a Miele G885 SC dishwasher after the power outage knocked out its software, the numerous problems that resulted in a worldwide recall of the luxury BMW 745i sedans, the theft of customer information due to security breaches of information systems, and the loss of life due to friendly fire by the Patriot missile defense system over the sky of Iraq. Software is everywhere in today's world, and software failure affects everybody. A study sponsored by the NIST in 2001 found that software errors cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product [1]. The software industry must improve the quality of their products now, or else a consumer revolt may force software developers to act responsibly. This position paper will first examine the meaning of trustworthy software and the challenges in developing such software. Then we will propose ways to improve the current industrial state-of-practice to enable the production of more trustworthy software.

2. What is trustworthy software?

The Webster's Dictionary defines *trustworthy* as *deserving of trust or confidence*, *dependable*, *reliable*. *Dependability*, on the other hand, was defined by Laprie as "*the trustworthiness of a computer system such that reliance can justifiably be placed on the service*", and the *dependability of a system* can be defined by a set of attributes that include *availability*, *reliability*, *safety*, *security* (*confidentiality* and *integrity*), and *maintainability* [2]. Hence, the term "trustworthy software" is synonymous with "dependable software," and software's trustworthiness is a composite vector of the software's dependability attributes.

One difficulty in defining trustworthy software, and the trustworthiness of any consumer product and service in general, is the fact that many of the "-ilities" are non-quantitative attributes. Consumers usually draw an opinion on the trustworthiness of a product or a service based on their own experiences or opinions of other users. Trustworthiness has multiple facets and users of the same product may assess the product's trustworthiness differently. For example, a daily commuter may find his automobile (and its embedded software) dependable and trustworthy because the automobile seldom breaks down during his commute from home to work everyday, while a professional auto-race driver would find the same automobile undependable and not trustworthy because it fails to deliver the explosive acceleration needed to overtake another car in the last lap of the race.

* This research is supported by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

3. Challenges in building trustworthy software

Many factors contribute to the difficulties in building trustworthy software. The increase demand for automation and services rendered by computing systems has resulted in very large and complex software. The desire to capture more market share in the business world often forces designers to face the difficult (if not impossible) task of reconciling a multitude of conflicting requirements and constraints in order to produce a "one-size-fits-all" software. The sheer size of the software makes the development of a trustworthy product very difficulty, and the "one-size-fits-all" nature of the software can invite "unintended" use of the product by the users that will eventually cause the software to fail.

With the advance in Internet and wireless technology, there is a growing demand for network centric applications that are made up of heterogeneous legacy systems interacting with one another via a middleware. Each of these legacy systems were originally designed for a different set of requirements in a different context. Even if the individual legacy systems were proven to be dependable and trustworthy, the integration of these systems provides ample opportunity for software problems, ranging from interoperability problems to network security and quality-of-service problems.

The new demand for high performance and intelligent automobiles, aircraft and autonomous robots has pushed the complexities of embedded systems to a new level. These systems now need to interact closely with other embedded systems and function under much tighter timing and control constraints. The need to interact closely with their environment also makes the understanding and satisfaction of their safety requirements (i.e. "what the software must not do") a number one priority.

4. Improving the Industrial State-of-Practice

While certification by trustworthy organizations like the Underwriter's Laboratory can help establish a product's trustworthiness, three criteria must be met before certification can be effective*:*

- the existence of quantifiable trustworthy attributes for the organization to measure;
- a widely accepted trustworthy standard (i.e. the set of trustworthy attributes and their acceptable values) for the organization to compare the software product against;
- high-quality software products that are measured up to the trustworthy standards.

Moreover, test and certification alone will not produce high quality products; it requires a culture where dependability and trustworthiness issues are taken seriously. We need new strategies and methods that are readily transferable to industry to develop software that are more trustworthy.

We need a holistic approach over the entire development process to establish trustworthy criteria (the set of trustworthy attributes and their relative importance) for every artifact and identify the trustworthiness-providing activities in every phase of the software development; see for instance the SQUALE framework proposed in [3].) In addition, we need to track the dependence of the artifacts and hence the correlations of their trustworthy criteria to maximize the overall effectiveness of the trustworthiness-providing activities and the trustworthiness of the final product; for example see the framework proposed in [4].

One way to improve on the current industrial state-of-practice is through the use of practical lightweight formal methods. For example, we present in [5] an iterative process for studying the temporal behavior of the system-of-systems architecture that combines use case analysis, modeling and simulation, and run-time verification of temporal assertions. The process started with Use Case analysis, which is a common approach practiced by the software industry. Based on the use cases, we developed an object-oriented architecture of the system expressed in UML-RT and

augmented the model with formal specification of timing requirements in terms of a time-series temporal logic. We refined the internal structures of the component systems using the HIPO technique until components were readily mapped to modules of the target simulation written in OMNeT++. The UML-RT models were translated into coarse-grained simulation models that are exercised using the OMNeT++ simulation engine. The temporal assertions were fed into the DBRover, a time-series temporal logic tool that builds and executes temporal rules for the target application [6]. The OMNeT++ simulation code is instrumented with probes (code snippets) to send information to the DBRover for temporal property verification during prototype execution.

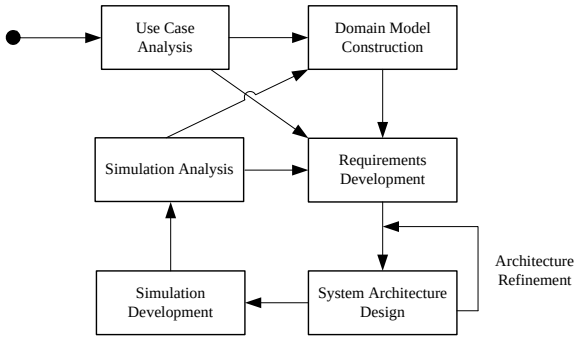

Figure 1. The Iterative Prototyping Process

.

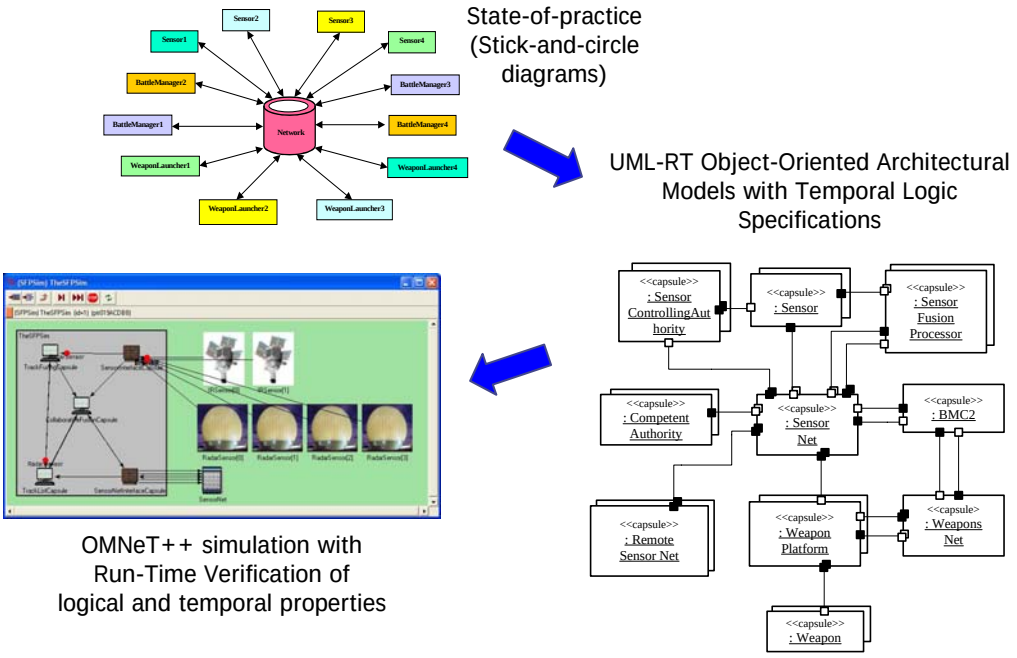

Figure 2. Improving the industrial state-of-practice with lightweight formal methods

5. Conclusion

Trustworthy software is an important enabling factor for technological changes. The software industry must improve their processes and methodologies to develop more dependable and trust-

worthy products. It will take a change of culture for the developers and consumers to take dependability and trustworthiness issues seriously. The improvements will come in small steps, with one improvement boot-strapping another towards the production of highly dependable and trustworthy software.

References

[1] RTI, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Planning Rpt. 02-3, National Institute of Standard and Technology, May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[2] J.C. Laprie (ed.), *Dependability: Basic Concepts and Terminology*, Springer Verlag, 1992.

[3] Y. Deswarte, "Dependable Computing System Evaluation Criteria: SQUALE Proposal", *Proc. Dependable Computing for Critical Applications 7*, June 1999, San Jose, pp. 317-400.

[4] J. Puett, Holistic *Framework For Establishing Interoperability of Heterogeneous Software Development Tools*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, Calif., June 2003.

[5] D. Drusinsky, J. B. Michael and M. Shing, "Behavioral Modeling and Run-Time Verification of System-of-Systems Architectural Requirements", to appear in the *Proc. CCCT'04*.

[6] D. Drusinsky, "The Temporal Rover and ATG Rover", *Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science*, 1885, pp. 323-329.

Can We Trust Intangibles?

Jeffrey Voas
Cigital, Inc.
voas@cigital.com

This position paper addresses aspects of the key questions of the workshop: (1) What does "trustworthy software" mean to you? and (2) How can a national software agenda be created to ensure that commercial software is of high integrity?

To begin, software, unlike hardware, is non-physical. You cannot smell it, see it, or touch it when it executes. Consequently, non-physical entities suffer from the inability to be easily measured, tested, or analyzed. However when considering physical systems, there are many more properties for which some sort of accurate measurement does exist (*e.g.*, by using an electron microscope).

Clearly we can measure code properties such as the number of statements or the complexity of the logic, but that is not a direct measure of how the software will behave at run-time. Those behaviors are much more difficult to predict, and can only be predicted if the environments around them are tightly bounded. This is something that occurs in the hardware domain, and therefore there are organizations such as Underwriter's Lab (UL) that test and accredit a wide variety of physical devices. The predictability variance between tangibles and intangibles is a key reason that software certification laboratories are not popping up all across the nation.

Further, software cannot be exhaustively tested. Limited testing is the norm, and the quality of that testing also greatly impacts our ability to predict and thus trust the software. (By quality, we mean the manner in which the test cases are selected. In software testing, more testing does not necessarily translate into more trust).

We also acknowledge that testing is only one family of techniques needed for trust. Other techniques include accurate requirements elicitation, tools of high integrity, complete and unambiguous requirements, proper assumptions about the mission, type of usage, and target environment. And since most systems live longer and longer lives, the role of accurate technology refresh during the maintenance of the system cannot be ignored if trust is to be maintained.

And finally, one piece of information often overlooked are the "negative requirements." These are the requirements that define what the software is *not* supposed to do. Without considering those during system conceptualization and architecture, it is possible that the system will be designed and deployed with those undesirable capabilities.

As for the second question of the workshop, it is clear that there is a need for a clearinghouse organization that can certify particular capabilities of software, whether COTS software or in-house software. Organizations like NASA have moved towards this by creating their own Independent Verification and Validation facility in Fairmont, WV. This is one step in that direction but not a complete step. And organizations such as Underwriter's Laboratory have created their own software safety standard for physical systems that contain software that could result in problems such as fire or electric shock (which are several of the main safety hazards that UL is on the lookout for). The fundamental problem with these approaches to date has not been a lack of standards, but a lack of ability to determine compliance and the fear of liability in providing guidance on how to comply. This has created a void in the trust of software, and that void has not even been alleviated by the open source movement.

Regardless of these issues, I believe that a "UL-like" organization can be created provided that the certificates that are generated (concerning how the software will behave and under what assumptions) are tightly bounded. By this, I mean instead of hoping for a certificate that states that the software "always works all of the time", begin with certificates that are limited in terms of: (1) what attribute (*e.g.*, reliability) and what functionality (*e.g.*, the ability to open a file) is being certified, (2) what environment, mission, and usage is assumed, and (3) what testing has been done to confirm the supporting data that is being used as evidence for the claims of the certificate (this is similar to making a safety case and similar to the evidence-claims-argument approach that is being discussed more frequently).

Thus I believe that if we are willing to limit the scope of the claims in a certificate to only that which is truly known, I believe we can generate believable and usable software certificates. And this can help greatly with problems related to component-based software and system-of-system interoperability.

In closing, I will leave with this 1998 quote from the National Academy of Science (which in my opinion is as valid today as it was in 1998):

> A consumer may not be able to assess accurately whether a particular drug is safe, but [they] can be reasonably confident that drugs obtained from approved sources have the endorsement of the U.S. Food and Drug Administration (FDA) which confers important safety information. Computer system trustworthiness has nothing comparable to the FDA. The problem is both the absence of standard metrics and a generally accepted organization that could conduct such assessments. There is no Consumer Reports for Trustworthiness**.References**

1. K. Miller and J. Voas. "An Ethical Can of Worms for Software Certifiers," In ***IT Pro***, September, 1(5): 18-20, 1999.
2. J. Voas. "Software Quality Tradeoffs, Return on Investment, and Software Safety," In ***Proceedings of the 20th International. System Safety Conference***, Denver, August, 2002.
3. J. Voas. "Certifying Off-the-Shelf Software Components," ***IEEE Computer***, 31(6): 53-59, June 1998. (Translated into Japanese and reprinted in Nikkei Computer magazine)
4. J. Voas. "Disposable Information Systems: The Future of Software Maintenance," ***Journal of Software Maintenance***, 11(2):143-150, March 1999.
5. J. Voas. "Certifying Off-the-Shelf Software Components," ***IEEE Computer***, 31(6): 53-59, June 1998. (Translated into Japanese and reprinted in Nikkei Computer magazine)
6. J. Voas. "Disposable Information Systems: The Future of Software Maintenance," ***Journal of Software Maintenance***, 11(2):143-150, March 1999.
7. J. Voas. "Certifying Software for High Assurance Environments," ***IEEE Software***, 16(4):48-54, July 1999.
8. J. Voas, Guest Editor, ***IEEE Software*** "Assuring Software Quality Assurance," 20(3):48-49, May, 2003.
9. "***Trust in Cyberspace***," National Academy of Sciences Report, National Academy Press, 1998.

TRUSTWORTHY COMPUTING

Feiyi Wang
MCNC Research & Development Institute
fwang2@mcnc.org

This position paper addresses two issues: first, what does "trustworthy software" mean to individual users; second, how can a national research agenda be created to ensure that commercial software is of higher integrity? Trust and its associated term of interest, trust worthy software, are indeed very elusive concepts. Trust by definition is "firm reliance on the integrity, ability, character of a person or thing". Trustworthy is something that warrants such reliance, in other words, it has to be earned in someway. Thus, there are two common cases we can label software to be trustworthy: one, I installed a piece of software which does what it says or what I had expected without a glitch, and I say the software is trustworthy. In another case, I bought *X* software from *Y* company. Company *X* has a reputable name, or referred to by another trustee *Z*, therefore, I label software *X* as trustworthy.

Note that in both cases, trustworthiness doesn't have much to do with security per se. To take the above argument a bit further for a formal definition, I would like to quote Matt Bishop in his book "Computer Security: Art and Science", which I generally agree. "An entity is *trustworthy* if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements. *Trust* is a measure of trustworthiness, relying on the evidence provided."

Credible evidence can come in various forms: one can follow a particular development methodology or paradigm (for example, open source development, and many eyeballs' theory), design analysis and formal verification methods; standard practice or the software can be measured against a particular certification process for an imposed completeness and rigor. Ultimately, the sufficiency of such evidence will be judged against the given requirements.

The challenges and difficulties arise when we try to archive trustworthiness in large-scale software development, where most of mission-critical applications fall. I echo the concerns that: each generation of hardware development allows us to create ever larger, more complex systems, but we are also increasingly defeated by that complexity. It couldn't have been said better than an open call from DARPA, "The raw power that has so seductively invited us to build systems of unprecedented size and complexity has led to systems that are dauntingly difficult to debug and test (thus stretching out delivery times), regularly fail in practices, and are increasingly vulnerable to attack." In short, it is my belief that we are building something we often don't have full grasp, or we think we have but actually we don't, which has eroded every corner of building next generation trustworthy computing software.

With that in mind, there are a lot of things we can do as part of a national research agenda. *First*, more long term research investment. I understand the argument and incentive for sponsoring research of "low hanging fruits" and quick turn around. However, I do believe that there are not silver bullets and shortcuts in addressing such a fundamental challenge. Every step of good, tangible progress requires serious effort and serious time. Often times, those seemingly hard paths one hesitates to take are precisely the right paths to follow. *Second*, we need better curricula in trust computing education. Take computer security course for example, due to their practical importance, topics like IPSEC, WEP, etc. are something that one can't miss, while other subjects such as Bell-Lapadula model, noninterference and policy composition, confinement problem, system perspective on information assurance, formal methods for verification, etc. become secon-

dary or ignored (omitted?) at all. Granted, this is a fairly casual observation and there are many good lectures out there, but it is the trend that concerns me. *Third*, we need more awareness and work on cost-benefit analysis, and this is paramount to commercial software development. One has to understand that every security solution, or to the large, trustworthy enhancement effort, comes with some forms of costs: performance, usability, financial, or a combination. To justify this cost, there has to be some well-received analyses or even quantifiable results as incentive.

Needless to say, these analyses and observations are quite coarse-grained, but I do hope this can prompt further discussions and be helpful in some ways to the workshop.

Developing Trustworthy Software by Embedding the Common Criteria into the Software Engineering Process

Meg Weinberg
Mitretek Systems
smw@mitretek.org

Increasing effort has been focused on independent security evaluations as a method for ensuring the trustworthiness of software products. While the Common Criteria (CC) international standard (ISO/IEC 15408) has become the premier security standard for specifying security requirements and evaluating products against these requirements. U.S. Government policies such as the National Security Telecommunications and Information Systems Security Policy (NSTISSP) No. 11 place overwhelming emphasis on CC certification as the single basis for assessing the ability of IT products to implement the specified security functionality.

While the CC standard provides a well-structured framework for specifying the security requirements to be satisfied by a software product, it is the direct, independent evaluation of software products against the CC requirements that is the primary method of assessing trustworthiness. At a fundamental level, the concept of independent evaluations contributes significantly to achieving trustworthiness by introducing a level of independent objectivity that compels developers to produce software products that are less vulnerable to attack and abuse.

However, those conducting the assessment of software products primarily rely upon evidence from the development process (e.g., design, test coverage, user guidance) to determine if the product satisfies the security requirements. Consequently, the realization of this vision is achievable only if the evidence submitted for evaluation is of sufficient quality and accuracy to permit evaluators to discover security vulnerabilities.

Since the CC standard is new to most software developers, products currently in evaluation have not considered the CC requirements until after product completion. As a result, development evidence necessary for the independent evaluation are commonly reverse engineered or existing documentation is augmented to describe security details. While this approach satisfies the CC requirements, it reveals a lack of emphasis on security during the engineering process and confirms security remains an afterthought rather than an integral part of software development.

In an effort to reduce the level of effort required by developers to achieve CC certification while simultaneously increasing the trustworthiness of products, we advocate that existing cost-effective software engineering processes be leveraged to support the CC evaluation process. In doing so, one should streamline the evaluation process, reduce CC evaluation preparation costs, and provide independent security evaluators with as much insight into the product as reasonably possible, with the aim of providing the evaluator with the a good opportunity to identify significant product vulnerabilities and provide feedback to the software developers on what and possibly how to improve a product's software trustworthiness.

Thinking Legally in Cyberspace

Thomas C. Wingfield
The Potomac Institute for Policy Studies
twingfield@PotomacInstitute.org

The moment an attorney begins to explain the legal minutiae of any field, particularly one as complex as cyberspace, technologists' eyes begin to glaze over. There is a (largely accurate) view that many topics matter only to lawyers, and that numerous doctrines and schemes never approach the real world of research, development, and operations.

There is, however, a legal cyber-framework which is comprehensible, authoritative, and relevant. The purpose of this brief paper is to present the broad strokes of this framework, to give non-attorneys "the Roman numerals of the outline" to enable scientists and engineers to spot issues, consult intelligently with lawyers, and, perhaps, distill these thoughts into their ongoing research.

To begin with, it is helpful to imagine a cyber intrusion as occurring within one or more of three legal regimes. These regimes are law enforcement, intelligence collection, and military action. Each of these has its own legal structure, its own methodology, and its own priorities. Any given intrusion surely triggers at least one of these regimes, but may invoke any two or even all three. Understanding their basic tenets, and the means by which they are deconflicted, would be a good day's work for any non-attorney.

Law Enforcement is the most ubiquitous of the three. It is based in the domestic law of the country which suffers the intrusion, and is intended to deal with intruders as individuals or, at most, small conspiracies of a few people. The law enforcement paradigm in the United States is one of patient, painstaking evidence collection under tight constitutional and statutory limitations. Presumptions favor the accused, and investigations are undertaken with an eye toward prosecution and incarceration. It may operate against U.S. citizens, non-citizens within U.S. control, or citizens of other countries having extradition agreements with the U.S. Such agreements are possible only when each state party has a domestic law criminalizing the same behavior.

The strengths of this approach are the clarity of its rules and (relative) certainty of its processes. It provides a highly transparent means by which to identify and sanction bad actors in cyberspace, and allows for highly individualized treatment in different cases. The weaknesses include the slow pace of investigation (especially at the federal level), the need to apprehend the accused, and the advantage to which an intruder may put the numerous limitations on government action against him.

Intelligence Collection is the second paradigm, and it is the simplest of the three. There is only one major test: Is the target a U.S. person? If the intruder is a U.S. citizen, a resident alien, or a member of an international group composed largely of Americans, then the U.S. intelligence community has virtually no power to collect against him. If, however, the target is a foreign national not falling into a protected category, there are remarkably few limitations on collection. The laws and directives governing the intelligence community are scrupulously followed, in large part because of the aggressive oversight maintained by the cleared members and staff of the House and Senate intelligence committees.

This paradigm offers great freedom for prompt and intrusive collection against those not protected by the U.S. Constitution. The primary aim under this approach, however, is not immediate resolution, but maximum information collection. An intruder may be left to his own devices in-

definitely, in the hopes of learning as much as possible about him. Questions would include: What information is he seeking? What techniques is he using? Who is working with him? What organization or government is supporting him? Once the purely intelligence-gathering portion of the operation is mature, it may even be possible to launch a counter-intelligence operation, feeding him false information to act as a tracer, or to alter the adversary's decision-making. The weakness here, then, is that an intruder may "get away with it" for quite some time, and the Big Picture of the intelligence community may rationally demand that a few systems be compromised to provide the information needed to protect the greater good.

The third approach is military action. Here, there are two basic bodies of law: the *jus ad bellum* ("law of conflict management"), which governs the decision of whether the incident in question (or the group of incidents being analyzed as a legal whole) constitutes a "use of force" or even an "armed attack" under international law, and, second, the *jus in bello* ("law of war"), providing the four principles governing operations after an "armed attack" has activated the inherent right of self-defense.

While not intrinsically complex, this area of the law is composed of a large number of relatively straightforward concepts. The decision of whether or not one is at war in cyberspace is most easily and authoritatively answered by employing the Schmitt analysis. While a full review of this process is beyond the scope of this paper, a brief description is in order. Until recently, the consensus among top legal scholars in the US and abroad was that a *quantitative* approach to cyberwarfare made the most sense. That is, when evaluating whether an information operation rose to the level of a use of force or armed attack, one should disregard the means and focus exclusively on the ends. Whether an oil refinery is set ablaze from a one-ton bomb or from a line of malicious computer code doesn't matter. A flaming refinery, they concluded, is a flaming refinery. Any cyber-attack that causes damage indistinguishable from a kinetic attack should be legally indistinguishable as well.

There is, unfortunately, a catch—the UN Charter, the paradigmatic document of international law, takes a *qualitative* approach, not a quantitative one. The framers, writing at the end of WWII, wanted to discourage military coercion, even at the cost of increasing diplomatic and economic coercion. Deciding that even the most stiffly worded diplomatic note—or restrictive economic boycott—would be preferable to an armored division crashing across an international border, the framers incorporated a very low threshold for impermissible military activity and a very high threshold for nonmilitary activity. The problem with this approach, as the subsequent decades have shown, is that many forms of "nonmilitary" coercion—such as terrorism and so called "low intensity conflicts"—result in more death and destruction than many traditional military activities, and many of today's information weapons look nothing like military weapons and technology of the past. Sixty years ago, a telegraph message was simply a means of communication, benign and unassuming. Perhaps today—and certainly in the future—its e-mail equivalent could carry a virus capable of wreaking just the sort of havoc described above.

Policy makers can overcome this intellectual and legal quandary by adhering to a forward-looking doctrine known as the "Schmitt Analysis." By demonstrating *how* military coercion differs from diplomatic and economic coercion, Michael Schmitt, late of Yale, the Naval War College, and now at the Marshall Center in Europe, identified seven areas—severity, immediacy, directness, invasiveness, measurability, presumptive legitimacy, and responsibility—in which military operations differ qualitatively from nonmilitary ones. If any given operation were quantitatively "graded" in each of these seven areas, the results could be used to give a principled qualitative description of the operation, accurately classifying it as a use of force or not. In short, the quan-

titative input suggested by the world's leading scholars on the topic would yield the qualitative output required for a proper characterization under the UN Charter.

This analysis has the added advantage of requiring attorneys and their clients to make conscious, documented assessments of each facet of a proposed operation, educating them in an otherwise subjective, opaque, and often incomplete *ad hoc* analysis. This is as true of planning an offensive operation as it is of analyzing, and reacting lawfully to, an attack in which we are the victims.

How specifically should decision-makers use this information to think more clearly about ordering such operations? Here are the dyads they should keep in mind:

- Severity: If people are killed or there is extensive property damage, the action is probably military; the less damage, the less likely the action is a "use of force."
- Immediacy: When the effects are seen within seconds to minutes—such as when a bomb explodes—the operation is probably military; if the effects take weeks or months to appear, it is more likely diplomatic or economic.
- Directness: If the action taken is the sole cause of the result, it is more likely to be viewed as a use of force; as the link between cause and effect attenuates, so does the military nature of the act.
- Invasiveness: A violated border is still an indicator of military operations; actions that are mounted from outside a target nation's borders are probably more diplomatic or economic.
- Measurability: If the effect can be quantified immediately—such as photographing a "smoking hole" where the target used to be—the operation has a strong military characteristic; the more subjective the process of evaluating the damage, the more diplomatic or economic.
- Presumptive Legitimacy: State actors have a monopoly on the legitimate use of kinetic force, while other non-kinetic actions—attacks through or in cyberspace— often are permissible in a wider set of circumstances; actions that have not been the sole province of nation-states are less likely to be viewed as military
- Responsibility: If a state takes visible responsibility for any destructive act, it is more likely to be categorized as a traditional military operation; ambiguous responsibility militates for a non-military label.

Once the determination has been made that a state of *de facto* hostilities do exist and the law of armed conflict does apply, the analysis turns to the four central principles of the law of war: discrimination, necessity, proportionality, and chivalry.

The principle of *discrimination* requires that, prior to any use of force, the attacker distinguish between combatants and noncombatants, between military objectives and civilian objects. Noncombatants include civilians and "protected persons," such as medics and chaplains. While it is unlawful to attack civilian objects as such, it is not unlawful to collaterally damage civilian objects located so close to lawful military objectives that no reasonable precautions could have prevented the damage. In cyberspace, such distinctions will require the collection and analysis of information to produce the intelligence required to make such determinations. The military commander must consider foreseeable secondary effects in his target selection and mission planning.

The principle of *necessity* permits the use of all force required for mission accomplishment and force protection, but not superfluous force or unnecessary suffering. Superfluous force is that which is inflicted solely for the sake of causing damage. Unnecessary suffering has two components: quantitatively, it is simply the complement of military necessity (necessary force); quali-

tatively, it includes those means and methods of war that are by nature inhumane (chemical weapons, biological weapons, exploding bullets, transparent bullets which cannot be detected in the body by x-ray equipment).

The principle of *proportionality* requires the military commander to balance the collateral damage (against civilians and their property) of a planned attack against the concrete and direct military advantage expected to be gained. Proportionality is *not* the requirement to respond with only as much force as was used in the precipitating attack, and is *not* the requirement to employ the same means and methods as in the provoking attack. Whatever other policy preferences may be expressed later in the national security decision-making process, the basic legal requirement is to ensure the balance of military advantage gained outweighs the damage done to noncombatants and their property.

The principle of *chivalry* permits ruses of war, but forbids perfidy. Perfidy is unlawfully deceiving an opponent that he is entitled to enjoy, or required to accord, protections of law, which are, in fact, inapplicable in the situation. Attacking an enemy facility from a vehicle displaying the Red Cross or the U.N. flag would be examples of perfidiously deceiving the enemy into according protections under law (not attacking the vehicle) which are not applicable. Broadcasting a false notice of cease-fire or calling for a *parlementaire* before attacking the unwary opponent would be an example of perfidiously deceiving one's opponent that he is entitled to a protection of law, which in fact, he is not. Ruses of war, on the other hand, are legal. They are wartime deceptions that convince an enemy to launch or withhold an attack based on tactical reasons, rather than perceived legal reasons.

There are other legal aspects of the military paradigm, worthy of a brief mention, if not full explanation, here. Treaty law reflects the four basic principles of discrimination, necessity, proportionality, and chivalry, described above, codifying and implementing customary international law. They occasionally lead international law by affirmatively proposing new norms and standards, at first applicable to only those nations which sign the agreements. Such new standards may become binding on other nations over time, if those other nations adopt the new norms, and do so out of a sense of legal obligation. Treaties applicable to an earlier age or a different type of warfare may nonetheless be useful in charting the course of the law of information conflict, to the extent that analogies may be drawn from them and applied to operations in cyberspace.

In addition to treaty law, there is another legal dimension, based on the various geographical regimes implicated by a contemplated course of action. The principle areas are the law of the sea, air law, space law, and foreign domestic (or host nation) law. The law of the sea addresses the metaphor of "innocent passage" in cyberspace, unauthorized broadcasting, and liability for damage done in wartime. Air law covers the metaphor of defining airspace (as applied to defining cyberspace), the legal restrictions on interfering with aircraft communications or navigation, as well as physical attacks on aircraft. Space law is useful in the distinction of air and space, the use of space platform for peaceful and non-peaceful purposes, and the liability of those who cause damage against or through the use of space platforms. Foreign domestic law is most applicable in the area of stationing arrangements and the limitations they place on the employment of weapons and forces forward deployed.

This completes a brief, informal tour of the law governing operations in cyberspace. A final point to remember is that any given intrusion may implicate more than one area of the law. For example: A stealthy but deep intrusion into a SCADA system at a nuclear power plant would first involve the FBI, following the law enforcement approach. As their investigators uncovered evidence of foreign involvement, the CIA, operating under the intelligence collection paradigm,

would begin tracing the action back to its origin overseas, learning as much as possible about the human and cyber network supporting it. Finally, if it were discovered that another government were responsible not only for the intrusion, but also for planting a cyber weapon which resulted in the release of substantial radiation, then a Schmitt analysis could determine the lawfulness of a military response. If the damage suffered by the U.S. were above the threshold of an "armed attack," then military action, either cyber or kinetic, could be conducted within the bounds of the four customary principles of the law of war.

The interaction of the law enforcement, intelligence collection, and military paradigms is an excellent basis for evaluating the legal consequences of any given intrusion in cyberspace. While the lawyers will handle the fine details of any such analysis, the overall legal structure is accessible to any scientist or engineer in the field. Mastering the legal fundamentals, and designing systems and defenses with them in mind, will go a long way toward building a more secure infrastructure, and a much safer future for the United States.

Trustworthy Software: What does it mean to me?

Debbie S. Wittmer
Naval Sea Systems Command
wittmerds@navsea.navy.mil

1. Background

Commercial-off-the-shelf (COTS) software products are increasingly being used in critical military systems. Marketplace considerations have changed dramatically over the past twenty-five years. Military systems have been historically designed and developed by the government with minimal industry performance flexibility due to hardware-centric systems and strict military standards applied. New market drivers are causing technology turnover every twelve to twenty-four months, at a time when military in-service systems service lives are being extended to twenty-plus years. In effect, COTS product life cycles and Military system life cycles are out of sync. Other considerations include: COTS products, primarily software, are designed and produced for environments more benign than Military system environments. COTS designs are controlled and owned by the vendors and their products are being integrated into systems-of-systems.

2. Discussion

There is no established and agreed upon formal and standardized methodology for military warfare system software measurement criteria, that when properly measured, a system can be certified with confidence as integrated, interoperable, safe, and effective for the warfighting mission. My question will always be, "How do you verify that the required performance at any measurable level is adequate to ensure the lowest acceptance of risk?

Military systems have been involved in a serial systems engineering process that take years of requirement analysis, architecture design, systems development, test and evaluation, system implementation, and maintenance. The technical sophistication has only increased in recent years to draw attention to primary operations and performance being software driven. The military "systems" community has to adapt to becoming a "software" community. The value of software credibility has yet to be recognized.

Key "high-level" criteria have been established to ensure operational performance can be characterized from software intensive systems within a military framework. It is known that the desired products, or system, meet these criteria in order to be approved for implementation. These criteria are subjective in nature and difficult to assess.

Government, industry, and commercial developers utilize various standards therefore, creating systems that are restricted to functional improvements without extensive redesign and/or require significant resources.

3. Synopsis

- Trustworthy software is mature software that demonstrates compliance with documented system integration, interoperability, safety, and security requirements.
- Trustworthy software requires government, industry, and commercial standard methodology with objective criteria when examined ensures utilization of any product in multiple environments.
- Trustworthy software is vital to moving forward and making progress in today and tomorrow's Department of Defense fighting needs.

The Impact of the Feature of Dynamic Interactions of Web Services on Software Trustworthiness

Jia Zhang
Northern Illinois University
jiazhang@cs.niu.edu

The concept of Web services opens a new way of engineering software to quickly develop and deploy Web applications, by integrating other software components that are published as Web services and are accessible via the Internet using standard protocols. On the horizon, one can imagine future software applications that have constant communications with and significant functionalities coming from external Web services.

As Web services have the potential to become a more significant reality of software development, perhaps it is important to ask how they effect the conversation concerning software trustworthiness.[9] The basic contention of this paper is that, due to the increased prevalence of dynamic external interactions, it may become important to increase the domain of study of software trustworthiness beyond solitary systems to include the interactions between these systems.

The first analogy one might draw with Web services is to current third-party libraries one might use in developing a software application. However, there are some fundamental differences between a Web service and a static third-party library, which makes this analogy not terribly useful. Namely, a Web service is potentially a dynamic entity controlled and hosted by another organization. There are no guarantees that the code underlying the Web service is not being updated; therefore, the inherent trustworthiness of the Web service may be varied with time. In this sense, we mean that these external interactions are dynamic.

Perhaps a more fruitful analogy for a Web service is a software application's interaction with a user. Certainly, a user's interaction with an application (particularly a database application) can affect its trustworthiness. However, given the fact that the people involved in an interaction can change or that a given person's trustworthiness could vary from day to day, there is a certain dynamic nature in a user's interaction that does not lend itself to static quantification.

When deciding the trustworthiness of a software product that relies on other Web services, the effects of external interactions to Web services need to be taken into account. In analyzing these dynamic external interactions, the first step is to try to quantify the trustworthiness impact they have on a given application. Seemingly there are three factors that would weigh on an interaction's trustworthiness impact.

- How important is the interaction?
- What is the trustworthiness of the Web services currently?
- What is the level of potential dynamism of the Web services (or perhaps what is the trustworthiness of the organization that is administrating the Web service)?

[9] Personally, I believe that it is probably possible to come up with a definition for trustworthiness that would give any given solitary system an exact trustworthiness factor; however, I also recognize that almost by definition it would be impossible to precisely determine what that factor is. For example, the number and nature of bugs in an application would have a direct impact on its trustworthiness; if I knew the number and nature of the bugs contained in the application allowing me to come up with an exact measurement of its trustworthiness, wouldn't I simply fix the bugs?

An external interaction with a Web service that was of little importance to an application would have little impact on that application's trustworthiness. Whereas an interaction that was essential to an application's functioning could have a significant affect on that application's trustworthiness.

The testing means and tools that are developed to quantify the trustworthiness of a static third-party library could be used to quantify the trustworthiness of an external Web service. Of course, one would have to be cognizant of the diminished helpfulness of that factor due to the service's dynamic nature.

Trying to rate the organization that is administrating a Web service could become an important step in determining one's own software's trustworthiness. Certainly an organization that is prone to deploying changes to their services frequently and with insufficient testing and notifications would have a tangible impact on the trustworthiness of any and all applications that relied on their services.

This process would perhaps create someway of quantifying the trustworthiness impact external interactions have with a given application. The question is that, just because a web service has a given importance and trustworthiness, is it possible to unilaterally increase the trustworthiness of such an interaction if its inherent trustworthiness is insufficient?

One fairly obvious answer is redundancy. Assume that an application requires high trustworthy service; and it is discovered that none of the Web services alone at one specific time can provide the requisite trustworthiness. The application could make use of two such services simultaneously. In this way, the external interactions become both more robust and more accurate.

Web services have the potential to become a significant phenomenon. They pose interesting and potentially difficult problems when trying to quantify their trustworthiness, as with any dynamic interactions an application may have, such as with a user. In the context of trustworthiness it will become important to first discover a way to quantify the trustworthiness impact of these external interactions. And then, perhaps, develop mechanisms to unilaterally improve the trustworthiness of these interactions.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Major General Alan B. Salisbury, USA (Ret)
   Center for National Software Studies

4. Peter J. Denning
   Naval Postgraduate School

5. James B. Michael
   Naval Postgraduate School

6. Jeffrey M. Voas
   Cigital Research Labs

7. Richard C. Linger
   Software Engineering Institute

8. Marshall D. Abrams
   The MITRE Corporation

9. William W. Agresti
   Johns Hopkins University

10. Larry Bernstein
    Stevens Institute of Technology

11. Valdis Berzins
    Naval Postgraduate School

12. Michael L. Brown
    Naval Postgraduate School

13. Dale S. Caffall
    Missile Defense Agency

14. LTC Thomas Cook, USA
    Missile Defense Agency

15. COL Kevin J. Greaney, USA
    Missile Defense Agency

16. Charles C. Howell
The MITRE Corporation

17. Ronald J. Kohl
R. J. Kohl & Associates, Inc.

18. T. Y. Lin
San Jose State University

19. Keith W. Miller
University of Illinois at Springfield

20. Peter G. Neumann
SRI International

21. Hal Pierson
Federal Aviation Administration

22. Arthur Pyster
Federal Aviation Administration

23. Samuel T. Redwine, Jr.
James Madison University

24. Norman F. Schneidewind
Naval Postgraduate School

25. Man-Tak Shing
Naval Postgraduate School

26. Feiyi Wang
MCNC Research & Development Institute

27. Sarah M. Weinberg
Mitretek Systems

28. Thomas C. Wingfield
Potomac Institute for Policy Studies

29. Debbie S. Wittmer
Naval Sea Systems Command

30. Jia Zhang
Northern Illinois University