NISTIR 8151

Dramatically Reducing Software Vulnerabilities

Report to the White House Office of Science and Technology Policy

Paul E. Black
Lee Badger
Barbara Guttman
Elizabeth Fong

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8151

National Institute of
Standards and Technology
U.S. Department of Commerce

NISTIR 8151

Dramatically Reducing Software Vulnerabilities

Report to the White House Office of Science and Technology Policy

Paul E. Black
Lee Badger
Barbara Guttman
Elizabeth Fong
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8151

November 2016

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

National Institute of Standards and Technology Interagency Report 8151
64 pages (November 2016)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8151

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at http://csrc.nist.gov/publications.

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Software and Systems Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8970) Gaithersburg, MD 20899-8970
Email: paul.black@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

The call for a dramatic reduction in software vulnerability is heard from multiple sources, recently from the February 2016 Federal Cybersecurity Research and Development Strategic Plan. This plan starts by describing well known risks: current systems perform increasingly vital tasks and are widely known to possess vulnerabilities. These vulnerabilities are often not easy to discover and difficult to correct. Cybersecurity has not kept pace, and the pace that is needed is rapidly accelerating. The goal of this report is to present a list of specific technical approaches that have the potential to make a dramatic difference in reducing vulnerabilities – by stopping them before they occur, by finding them before they are exploited or by reducing their impact.

Keywords:

Measurement; metrics; software assurance; software measures, security vulnerabilities; reduce software vulnerability.

Acknowledgements:

Many thanks to Rajeev Joshi (rajeev.joshi@jpl.nasa.gov) for contributions to Sect. 2.3 Additive Software Analysis Techniques.

Thanks also to W. Konrad Vesey (william.k.vesey.ctr@mail.mil), Contractor, MIT Lincoln Laboratory, Office of the Assistant Secretary of Defense, Research and Engineering, for material in Sect. 2.5 Moving Target Defense (MTD) and Automatic Software Diversity. Much of the wording is directly from a private communication from him.

We thank Terry Cohen, Mark Cornwell, John Diamant, Jeremy Epstein, D. Richard Kuhn, Andrew Murren, Kenneth S Thompson, Jan Vandenbos, David Wheeler and Lok Yan for their significant comments and suggestions, which greatly improved this report. We also thank the many others who suggested improvements, asked questions and submitted comments.

Table of Contents

1 Introduction 1
1.1 Scope of Report 2
1.2 Findings 3
1.3 Audience 4
1.4 Measures 4
1.5 Methodology 4
1.6 Report Organization 5
2 Technical Approaches 5
2.1 Formal Methods 6
2.1.1 Sound Static Program Analysis 7
2.1.2 Model Checkers, SAT Solvers and Other “Light Weight” Decision Algorithms .. 8
2.1.3 Assertions, Pre- and Postconditions, Invariants, Aspects and Contracts 8
2.1.4 Correct-by-Construction and Model-Based Development 9
2.1.5 Directory of Verified Tools and Verified Code 9
2.1.6 Cyber Retrofitting: Putting Formal Methods to Work 10
2.2 System Level Security 11
2.2.1 Operating System Containers 13
2.2.2 Microservices 13
2.3 Additive Software Analysis Techniques 16
2.3.1 Software Information Expression and Exchange Standards 16
2.3.2 Tool Development Framework or Architecture 18
2.3.3 Strategy to Combine Analysis Results 18
2.3.4 Technology to Combine Analysis Results 19
2.4 More Mature Domain-Specific Software Development Frameworks 21
2.4.1 Rapid Framework Adoption 24
2.4.2 Advanced Test Methods 24
2.4.3 Conflict Resolution in Multi-Framework Composition 25
2.5 Moving Target Defenses (MTD) and Automatic Software Diversity 26
2.5.1 Compile-Time Techniques 26
2.5.2 System or Network Techniques 27

2.5.3 Operating System Interface Techniques 27
3 Measures and Metrics 29
3.1 A Taxonomy of Software Measures 30
3.2 Software Assurance: The Object of Software Measures 32
3.3 Software Metrology 33
3.4 Product Measures 34
3.4.1 Existing Measures 35
3.4.2 Better Code 35
3.4.3 Measures of Binaries and Executables 36
3.4.4 More Useful Tool Outputs 36
3.5 Further Reading 36
4 Non-Technical Approaches and Summary 38
4.1 Engaging the Research Community 38
4.1.1 Grand Challenges, Prizes and Awards 38
4.1.2 Research Infrastructure 39
4.2 Education and Training 39
4.3 Consumer-Enabling Technology Transfer 41
4.3.1 Contracting and Procurement 41
4.3.2 Liability 42
4.3.3 Insurance 42
4.3.4 Vendor-Customer Relations 42
4.3.5 Standards 42
4.3.6 Testing and Code Repositories 43
4.3.7 Threat Analysis 43
4.4 Conclusions 43
4.5 Table of Acronyms 45
5 References 47

1 Introduction

The call for a dramatic reduction in software vulnerability is being heard from multiple sources, including the February 2016 Federal Cybersecurity Research and Development Strategic Plan [FCRDSP16]. This plan starts by describing a well-known risk: current systems perform increasingly vital tasks and are widely known to possess vulnerabilities. These vulnerabilities are often not easy to discover and difficult to correct. Cybersecurity has not kept pace, and the pace that is needed is rapidly accelerating. The plan defines goals for the near, mid and long term. This report addresses the first mid-term goal:

> *Achieve S&T [Science and Technology] advances to reverse adversaries' asymmetrical advantages, through sustainably secure systems development and operation. … This goal is two-pronged: first, the design and implementation of software, firmware, and hardware that are highly resistant to malicious cyber activities (e.g., software defects, which are common, give rise to many vulnerabilities) …*

There are many different definitions of the term "vulnerability" covering various combinations of concepts, including knowledge, attacks, exploitability, risk, intention, threat, scope and time of introduction. For the purposes of this report, we define *vulnerability* as one or more weaknesses that can be accidentally triggered or intentionally exploited and result in a violation of desired system properties. A weakness is an undesired characteristic of a system's requirements, design or implementation [Black11a]. This definition excludes

- manual configuration or operational mistakes, such as installing a program as world-readable or setting a trivial password for administrator access;
- insider malfeasance, such as exfiltration by Edward Snowden;
- functional bugs, such as the mixture of SI (International System of Units) and Imperial units, which led to the loss of the Mars Climate Orbiter in 1999 [Oberg99];
- purposely introduced malware or corrupting "mis-features" in regular code, such as allowing root access if the user name is "JoshuaCaleb" and
- software weaknesses that cannot be exploited (by "outsiders") as a result of input filtering or other mitigations.

Great strides have been made in defining software vulnerabilities, cataloging them and understanding them. Additionally, great strides have been made in educating the software community about vulnerabilities, attendant patches and underlying weaknesses. This work, however, is insufficient. Significant vulnerabilities are found routinely, many vulnerabilities lie undiscovered for years and patches are often not applied. Clearly a different approach—one that relies on improving software—is needed.

> *Strengthening protection requires increasing assurance that the products people develop and deploy are highly resistant to malicious cyber activities, because they include very few vulnerabilities …* [FCRDSP16, p. 17]

1.1 Scope of Report

The goal of this report is to present a list of specific technical approaches that have the potential to make a dramatic difference reducing vulnerabilities – by stopping them before they occur, by finding them before they are exploited or by reducing their impact.

- Stopping vulnerabilities before they occur generally includes improved methods for specifying, designing and building software.
- Finding vulnerability includes better testing techniques and more efficient use of multiple testing methods.
- Reducing the impact of vulnerabilities refers to techniques to build architectures that are more resilient, so that vulnerabilities cannot be exploited for significant damage.

The report does not segregate the approaches into these three bins, since some approaches may include pieces from multiple bins.

The list of approaches in this report is not comprehensive. It is intended to show how a wide variety of approaches can make a significant impact. New approaches will continue to be developed and brought to general use.

The list of approaches for reducing vulnerabilities focuses on approaches that meet three criteria:

1. Dramatic impact,
2. 3 to 7-year time frame and
3. Technical activities.

Dramatic. This means approaches that are broadly applicable and that significantly contribute to the goal of reducing vulnerabilities by two orders of magnitude. In the case of typical software, estimates are up to 25 errors per 1000 lines of code [McConnell04, p. 521]. Nearly two-thirds of vulnerabilities come from simple programming errors [Heffley04]. These approaches have been selected for the possibility of reinforcing the ambition of reaching 25 errors per 100 000 lines of code for those types of software and achieving corresponding reductions for other types. (Systems with near-zero errors are produced routinely today in the aerospace industry, but at several times the cost of ordinary software.) Determining whether an approach has a dramatic impact requires the ability to measure it. Measuring software quality is a difficult task. Parallel efforts on improving the measurement of software vulnerabilities are going on.

3- to 7-year time frame. This time frame was selected because it is far enough out to make dramatic changes, based on existing techniques that have not reached their full potential for impact. It is a time frame that it is reasonable to speculate about. Beyond this time frame, it is too difficult to predict what new technologies and techniques will be developed, potentially making their own set of dramatic changes on how information technology is used. In the near future, the emphasis will be on implementing techniques that are already being deployed, such as work in secure software development and testing. The dividing line between deployed and future techniques is not crisp. If a technique is widely used or used by major software developers, it was not included although broader adoption of the technique would be beneficial.

Technical. There are many different types of approaches to reducing software vulnerabilities, many of which are not primarily technical – from helping users meaningfully request security to funding research and operational activities and training all parties, who design, build, test and use software. During the development of this report, many ideas were put forward across this broad span. The report only addresses technical approaches in order to have a manageable scope, which builds on expertise available during the development of the report. These other areas are critical, too.

During the drafting of this report, many excellent ideas were brought forth that are outside the scope of this report and are summarized in Section 4. Examples of these activities include:

- Improved funding,
- Improving education,
- More research for various aspects of software understanding,
- Increased use of grand challenges and competitions,
- Providing better methods for consumers of software to ask for and evaluate lower-vulnerability software,
- Liability and standards and
- Threat analysis.

This report excludes a discussion of vulnerabilities in hardware. This is not to say that these are not critical. These can be addressed in another report. This report targets a broad range of software, including government-contracted software, proprietary and open source software. It covers software for general use, mobile devices and embedding in appliances and devices. The goal is to prevent vulnerabilities in new code, in addition to identifying and fixing vulnerabilities in existing code.

1.2 Findings

While the problem of reducing software vulnerabilities is a difficult one and clearly requires a mixture of technical, operational, managerial, psychological and cultural changes to address the problem, it is possible to make a dramatic difference. This report describes five mid-term approaches that have the potential to address technical aspects of the problem. The approaches included here are not a comprehensive list; they represent a wide range of potential approaches and highlight how reducing software vulnerabilities can be accomplished. All of these approaches will require improved research infrastructure, including significantly better metrics.

As noted, they cannot be successful by themselves and will need to be integrated into the larger software developer and user communities. Furthermore, this report does not focus on current trends in secure software development that are already in significant use.

1.3 Audience

The primary audience for the report is the White House Office of Science and Technology Policy (OSTP). It is anticipated that other policy bodies, funders and researchers will find the report useful as they consider investments and programs to improve software quality. Since the report focuses on a three to seven-year time frame, it is not intended as a guide for software developers.

1.4 Measures

There are multiple efforts to define software vulnerabilities, their prevalence, their detectability and the efficacy of detection and mitigation techniques. The ability to measure software can play an important role in dramatically reducing software vulnerabilities. Industry requires evidence of the extent of such vulnerabilities, in addition to knowledge in determining which techniques are most effective in developing software with far fewer vulnerabilities. With effective measures that can function as market signals, industry can favor and select low vulnerability software, thus encouraging development of better software [Grigg08]. Additionally, and more critically, industry requires guidance in identifying the best places in code to deploy mitigations or other actions. This evidence comes from measuring, in the broadest sense, or assessing the properties of software.

1.5 Methodology

In order to produce the list of approaches, the Office of Science and Technology Policy (OSTP) asked NIST to lead a community-based effort. The report was developed during an eight-month period. Given the compressed time frame, the focus of the report was kept to the criteria described above to highlight promising approaches rather than perform a comprehensive analysis. NIST consulted with multiple experts in the software assurance community including:

- Two OSTP-hosted inter-agency roundtables;
- Half-day session at the Software and Supply Chain Assurance (SSCA) Summer Forum;
- All-day workshop on Software Measures and Metrics to Reduce Security Vulnerabilities;
- Two-day workshop on Reducing Software Defects and Vulnerabilities, hosted by the Software Productivity, Sustainability, and Quality (SPSQ) Working Group of the Networking and Information Technology Research and Development (NITRD) Program and
- Public comment from 4 to 18 October 2016.

1.6 Report Organization

The report is organized into two major sections. The first, Section 2, enumerates technical approaches and the second, Section 3, addresses measures.

Section 2 has subsections for technical approaches to deal with vulnerabilities in software. These include formal methods, such as sound static program analyses, model checkers and Boolean Satisfiability (SAT) solvers. It also suggests having a directory of verified tools and verified code. This section addresses system level security, including operating system containers and microservices. Additive software analysis techniques are addressed. Finally, it discusses moving target defenses (MTD) and automatic software diversity. These include compile-time techniques, system or network techniques and operating system techniques.

Each subsection follows the same format:

- Definition and Background: Definition of the area and background;
- Maturity Level: How mature the area is, including a discussion of whether the approach has been used in the “real world” or just in a laboratory and issues related to scalability and usability;
- Basis for Confidence: Rationale for why this could work;
- Rationale for Potential Impact and
- Further Reading, including papers and other materials.

Section 3 covers software measures. It is designed to encourage the adoption of measurement and tools to address vulnerabilities in software. It addresses product measures and how to develop better code. It also addresses the criticality of software security and quality measures.

After those two major sections are Section 4, on crosscutting issues, such as engaging the research community, education and vehicles to transition the technical approaches to general use, and references in Section 5.

2 Technical Approaches

There are many approaches at varying levels of maturity that show great promise for reducing the number of vulnerabilities in software. This report highlights five of them that are sufficiently mature and have shown success, so that it is possible to extrapolate into a three to seven-year horizon. This list is not meant to be exhaustive, but rather to show that it is possible to make significant progress in reducing vulnerabilities and to lay out paths to achieve this ambitious goal. One of the significant themes of the SPSQ workshop was the need to improve not just software, but also testing tools by applying formal techniques.

2.1 Formal Methods

Formal methods include all software analysis approaches based on mathematics and logic, including parsing, type checking, correctness proofs, model-based development and correct-by-construction. Formal methods can help software developers achieve greater assurance that entire classes of vulnerabilities are absent and can also help reduce unpredictable cycles of expensive testing and bug fixing.

In the early days of programming, some practitioners proved the correctness of their programs. That is, given language semantics, they logically proved that their program had certain properties or gave certain results. As the use of software exploded and programs grew so large that purely manual proofs were infeasible, formal correctness arguments lost favor. In recent decades, developments, such as the breathtaking increase in processing capacity predicted by Moore’s law, multi-core processors and cloud computing, made orders of magnitude more computing power readily available. Advances in algorithms for solving Boolean Satisfiability (SAT) problems, satisfiability modulo theories (SMT) [Bjørner16], decision procedures (e.g., ordered binary decision diagrams - OBDD) and reasoning models (e.g., abstract interpretation and separation logic) dramatically slashed resources required to answer questions about software.

One early effort at using formal methods to achieve far fewer vulnerabilities was the 1980s’ DoD Trusted Computer Security Evaluation Criteria (TCSEC). The TCSEC specified multiple levels of software assurance. The highest level, A1, required formal specification of the system and mathematical proof of correspondence between the code and the specification. Successful theorem proving tools were developed and several formally-proved systems were produced, but the expense and time required was prohibitive—as much as two years—for practical use.

By the 1990s, formal methods had developed a reputation as taking far too long, in machine time, person years and project time, and requiring a PhD in computer science and mathematics to use them. This is no longer the case. Formal methods are widely used today. For instance, compilers use SAT solvers to allocate registers and optimize code. Operating systems use algorithms formally guaranteed to avoid deadlock. Kiniry and Zimmerman call these “Secret Ninja Formal Methods” [Kiniry08]: they are invisible to the user, except to report that something is not right. In contrast to such “invisible” use of formal methods, overt use often requires recasting problems into a form compatible with formal methods tools.

Overt formal methods are recommended in automotive [ISO26262-6] and railway [Boulanger15] standards. Formal proof techniques have significantly reduced the effort to achieve objectives defined by the airborne standard, DO 178B [Randimbivololona99]. Its successor, DO 178C, has an entire supplement, DO 333, devoted to the use of formal methods for software verification. Most proposed cryptographic protocols are now examined with model checkers for possible exploits, and analysts can perform mostly-automated proofs that implementations of cryptographic algorithms match specifications [Carter13]. Practitioners also use model checkers to look for attack paths in networks.

Despite their strengths, formal methods are less effective if there is no clear statement of software requirements or if what constitutes proper software behavior can only be determined by human judgment or through balancing many conflicting factors. Thus, we would not expect formal methods to contribute as much to the evaluation of the usability of a user interface, development of exploratory software or unstructured problems.

Formal methods include many, many techniques at all stages of software development and in many different application areas. We do not list every potentially helpful formal method. Instead, we concentrate on a few methods that may contribute significantly in the mid-term.

2.1.1 Sound Static Program Analysis

Static analysis is the examination of software for specific properties without executing it. For our purposes, we only consider automated analysis. Heuristic analysis is faster than sound analysis, but lacks assurance that comes from a chain of logical reasoning. Some questions can only be answered by running the software under analysis, i.e., through dynamic analysis. Combining static and dynamic analysis yields a hybrid technique. In particular, executions may produce existence proofs of properties that cannot be confirmed using static techniques only.

Many representations of software (e.g., requirements, architecture, source code and executables) may be statically analyzed. Source code analysis, however, is the most mature. One advantage of source code analysis is that the context of problems identified in source code can be communicated to software developers using a familiar representation: the code itself. When other representations are analyzed, an additional step is required to render a warning into a form that people can first understand and then relate to a program under analysis.

According to Doyle’s assessment, sound static analysis is superior to current software development practices in terms of coverage, scalability and benefit for the effort [Doyle16]. We believe that one limitation is that it is difficult to specify some properties in available terms.

Formal specification and sound static analysis have shown significant applicability in recent years. For example, the Tokeneer project shows that software can be developed with formal methods faster and cheaper and with fewer bugs than with traditional software development techniques [Barnes06, Woodcock10]. TrustInSoft used Frama-C to prove the absence of a set of Common Weakness Enumeration (CWE) classes in PolarSSL, now known as mbed TLS [Bakker14, Regehr15]. Ourghanlian compares the use of PolySpace Verifier, Frama-C and Astrée to assess safety-critical software in nuclear power plants [Ourghanlian14]. Sound static analysis and other formal methods are extensively employed for software development in areas beyond transportation, aerospace and nuclear plant control [Voas16b].

These developments illustrate a few of the many uses of static analysis. Going forward, static analysis has the potential to efficiently preclude several classes of errors in newly-developed software and to reduce the uncertainty regarding resources needed to reach higher levels of assurance through testing.

2.1.2 Model Checkers, SAT Solvers and Other "Light Weight" Decision Algorithms

These algorithms can answer questions about desirable higher level properties, such as that a protocol only allows sensitive text to be read if one has a key, that security properties are preserved by the system, that an assignment of values satisfies multiple constraints or that there are no paths to breaches via (known) attacks. These algorithms can also be applied to analyze detailed design artifacts, such as finite (and infinite) state machines.

Doyle's assessment is that model checkers can have excellent coverage and many properties can be represented [Doyle16]. However, since the effort required increases exponentially with problem size, there is always an effectize size limit. Problems smaller than the limit can be solved quickly. Very large problems may require excessive resources or intensive human work to break the problem into reasonable pieces.

Such techniques can be applied in essentially two ways. First, they can be used as part of software in production. For instance, instead of an *ad hoc* routine to find an efficient route for a delivery truck, an application can use a well-studied Traveling Salesman or spanning tree algorithm. Second, and perhaps more pertinent to the theme of this report, is to use the algorithms to design or verify software.

2.1.3 Assertions, Pre- and Postconditions, Invariants, Aspects and Contracts

Programmers generally have a body of information that gives them confidence that software will perform as expected. A neglected part of formal methods is to unambiguously record such insights. Variants go by different terms, such as contracts, assertions, preconditions, annotations, postconditions and invariants. It may cost programmers some extra thought to state exactly what is going on using a language similar to code expressions, but such statements help. Automated aids, such as Counterexample-Guided Abstraction Refinement (CEGAR), can help produce statements. These statements are activated ("compiled in") during development and testing, then may be deactivated before release.

The benefit is that these formal statements of properties carried in the code may be used to cross-check the code. For example, tests may be generated directly from assertions. They may be activated to perform internal consistency checks during testing or production. Faults can therefore be detected much earlier and closer to erroneous code, instead of having to trace back from externally visible system failures. Assertion-based testing can detect up to 90 % of errors with an appropriate level of coverage of the input space [duBousquet04]. Such statements also supply additional information to perform semi-automated proofs of program correctness. Unlike comments, which may not be updated when the code changes, these can be substantiated or enforced by a computer and, therefore, must continue to be accurate statements of program features and attributes.

A striking example of how such formal statements could help is the 1996 failure of the first Ariane 5 rocket launched. The Ariane 5 used software from the successful Ariane 4. When the

Ariane 4 was designed, analysis showed that a 16-bit integer could handle its speeds. However, higher Ariane 5 speed values overflowed the variable, leading to computer shut down and the loss of the vehicle. If the code had a precondition that the speed must fit in a 16-bit integer, "Any team worth its salt would have checked … [preconditions, which] would have immediately revealed that the Ariane 5 calling software did not meet the expectation of the Ariane 4 routines that it called." [Jézéquel97]

2.1.4 Correct-by-Construction and Model-Based Development

In model-based development, a software developer creates and modifies a model of a system. Behavior may be specified in a higher-level or domain-specific language or model, and then code is automatically generated. Much or all of the code is generated from the model. This is one correct-by-construction technique. This technique and others, such as design by refinement, aim to avoid whole classes of vulnerabilities entirely, since the developer rarely touches the code. Code synthesis like this is useful in fewer situations than other formal methods, since it may be impractical to develop the modeling superstructure and code generator for an area, e.g., a user interface with error recovery and help prompts. Such models or specifications may also generate test suites or oracles. They may also be used to validate or monitor system operation.

When analysts can specify complete high-level models for entire systems, or even subsystems, we call the model a "domain-specific language" (DSL) and cease to consider it noteworthy. This represents a substantial use of formal methods. According to Doyle's assessment, program synthesis has an "A+" in coverage and "B" in effort and properties [Doyle16].

2.1.5 Directory of Verified Tools and Verified Code

Software developers often must expend significant effort to qualify tools or develop program libraries with proven properties. Even when a later developer wishes to use the results of such work, there are no central clearing houses to consult. A list of verified tools, carefully constructed libraries and even reusable specifications and requirements can speed the adoption of formal methods. Such a tool library could facilitate wider use, with accompanying assurance, of software with dramatically reduced numbers of vulnerabilities.

Many companies and government agencies evaluate the same tools or the same software for similar uses. Since it may be difficult to find out who may have done related evaluations, each entity must duplicate the work, sometimes with less knowledge and care than another has already applied. It is especially challenging since many contracts discourage sharing results [Klass16]. A repository or list would be of great benefit. Knowing about related efforts, developers could contribute to one effort, instead of working on their own. As a code or "live" instantiation of a repository, the Open Web Application Security Project (OWASP) foundation coordinated a project to develop a shared application program interface (API) that encapsulated key security operation, called Enterprise Security API (ESAPI).

See Sect. 2.4 for a discussion of re-use of well-tested and well-analyzed code.

2.1.6 Cyber Retrofitting: Putting Formal Methods to Work

The preceding techniques can be put to use to develop new software, however reworking all legacy software is impractical. This is called "cyber retrofitting" in analogy to seismic retrofitting for greater earthquake resistance [Fisher16]. The first step is to identify the most critical, key or foundational components in existing systems. SPSQ workshop participants also emphasized the use of formal methods for key modules rather than entire applications. A key component may be a compiler or run-time library, in addition to purpose-written code. These components are then closely examined or reengineered with the appropriate formal methods to gain higher levels of assurance, as was done for PolarSSL, now known as mbed TLS, [Bakker14, Regehr15] in the wake of Heartbleed or the CompCert Verifying C Compiler [Leroy06].

Another small step in cyber retrofitting is to recompile components with automatic strengthening or hardening. For instance, a compiler can add bounds checks where possible on all questionable memory accesses to greatly reduce the number of unrecognized buffer overflows (BOF class) [Bojanova16]. Such strengthening typically has little performance impact [Flater15]. Compilers also can replace typical unsafe functions with safer "fortified" functions and can employ executable space protection, i.e., mark memory regions for data as non-executable.

2.1.7 Maturity Level

Today formal methods are used, often relatively invisibly, throughout the world. One of the most pervasive applications is the use of strong type checking, which is a formal method, within modern programming languages. Other, admittedly limited, uses are the algorithms of various software checking tools, some of them built into widely used development environments (e.g., that tag inconsistent use of variables, missing values or use of unsafe interfaces). In 2010, researchers at National ICT (Information Communications Technology) Australia (NICTA) demonstrated the formal verification of the seL4 microkernel comprising about 10 000 lines of C code [Klein14]. The UK National Air Traffic Service (NATS) interim Future Area Controls Tools Support (iFACTS) has over 200 000 lines of SPARK source code, and "all code changes *must* prove OK before code can be committed" [Chapman14].

2.1.8 Basis for Confidence

Assertions, contracts, invariants and other formal declarations have been significantly adopted in high-quality software. Their gradual improvement to encompass more advanced conditions and API checking is because they have already proven themselves in developer communities. For instance, Source code Annotation Language (SAL 2.0) is available in Visual Studio 2015. Many tools now perform static analysis. A natural progression is to promote more and more advanced forms of static and hybrid analysis. Software proving based on techniques such as pre- and post-condition satisfaction and proof-carrying code have seen initial adoption in critical software [Souyris09]. They require more effort and cost, however, in some cases they have been shown to be cost effective in the long run: improvement in development time and cost in one-third of uses and fewer or no fixes to deployed systems [Woodcock09].

2.1.9 Rationale for Potential Impact

The greatest potential impact is likely in costs avoided for components that, over time, become heavily relied upon. The Heartbleed debacle is an example of a modest code base with outsized importance: a judicious use of formal methods might have avoided the problem in the first place. Generally, higher quality software, such as can be produced using formal methods, can be used to lower long-term maintenance and replacement costs of software components. Unlike physical systems that wear out and eventually fail, software systems suffer failures when they are incorrect and the flaws are triggered by environmental factors, such as, particular sequences or combinations of inputs [Woody14].

2.1.10 Further Reading

[Armstrong14] Robert C. Armstrong, Ratish J. Punnoose, Matthew H. Wong and Jackson R. Mayo, "Survey of Existing Tools for Formal Verification," Sandia National Laboratories Report SAND2014-20533, December 2014. Available: http://prod.sandia.gov/techlib/access-control.cgi/2014/1420533.pdf Accessed 12 October 2016.

[Chapman14] Roderick Chapman and Florian Schanda, "Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK," in *Proc. Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014*. Gerwin Klein and Ruben Gamboa, Eds., *Lecture Notes in Computer Science*, Vol. 8558, Springer, 2014, pp. 17-26, https://doi.org/10.1007/978-3-319-08970-6_2.

[Voas16a] Jeffrey Voas and Kim Schaffer, "Insights on Formal Methods in Cybersecurity," *IEEE Computer*, Vol. 49, Issue 5, May 2016, pp. 102–105, https://doi.org/10.1109/MC.2016.131. The positions of seven experts on formal methods.

[Voas16b] Jeffrey Voas and Kim Schaffer, "What Happened to Formal Methods for Security?", *IEEE Computer*, Vol. 49, Issue 8, August 2016, pp. 70-79, https://doi.org/10.1109/MC.2016.228. A follow-up roundtable about formal methods with those seven experts.

[Woodcock09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui and John Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, Vol. 41, Issue 4, October 2009, pp. 1-36, https://doi.org/10.1145/1592434.1592436.

2.2 System Level Security

When software is executed, the system context for the running software defines the resources available to the software, the APIs needed to access those resources and how the software may access (and be accessed by) outside entities. These aspects of a system context may strongly affect the likelihood that software contains vulnerabilities (e.g., complex or buggy APIs increase the likelihood), the feasibility of an attacker exploiting vulnerabilities (e.g., more feasible if system services are reachable from outside) and the impact an attack could have (e.g., both damage to system resources and mission-specific costs).

A long-standing goal of system designers is to build systems that are resistant to attack and that enforce desirable security policies on both programs and users. Started in 1965, the Multics system [Corbato65] combined a number of ideas (e.g., virtual memory, multi-processing and memory segments) to implement a computing utility that could protect information from unauthorized access. Starting in the 1970s, a number of security policy models were introduced to formalize the security responsibilities of the system layer. In 1976, the Bell-LaPadula (BLP) model [Bell76] provided a formal expression of mandatory security for protecting classified information: the BLP model allowed "high" (e.g., SECRET) processes access to "low" (e.g., UNCLASSIFIED) information for usability but prevented "low" processes from accessing "high" information. Goguen and Meseguer's noninterference model accounted for indirect information flows, also known as covert channels [Goguen84]. Biba's integrity model expressed mandatory security for integrity: it prevented possibly-malicious (low-integrity) data from being observed by high-integrity processes, thus reducing the risk that high-integrity processing and data might become corrupted [Biba77]. Boebert and Kain's type enforcement model provided a table-based access control mechanism to allow data to be transformed only by pre-approved programs [Boebert85]. These models provided necessary clarity regarding desirable security properties, but using the models in real-scale systems posed usability problems for system administrators, and software implementations of the models still contained exploitable flaws.

In 1999, DARPA started the Intrusion Tolerant Systems (ITS) program predicated on the notion that systems can be built to operate through, or "tolerate," even successful attacks. A number of other research programs followed that built on this idea [Tolerant07]. Essential concepts explored by these programs included the structuring of systems with redundant and diverse components unlikely to all be subverted by a single vulnerability, the introduction of new policy-enforcing software layers and the use of diagnostic reasoning components for automated recovery. The DARPA research thrust in tolerant systems recognized that the elimination of all vulnerabilities from real-world systems is an unlikely achievement for the foreseeable future. The research demonstrated substantial tolerance in red team testing (e.g., see [Chong05]), but the approaches also imposed significant configuration complexity, reduced execution speed and significantly increased resource (CPU, memory, etc.) requirements.

Recent advances, both in hardware and software, raise the possibility of developing security-enforcing and intrusion tolerant systems that are both performance and cost effective. Such systems have the potential to suppress the harm that software vulnerabilities can cause. On the hardware side, the low cost multicore and system-on-a-chip processors are lowering the costs of redundancy. On the complementary software side, emerging architectural patterns are offering a new opportunity to build security and tolerance into the next generation of systems. Among numerous possible patterns, two that appear promising are operating system containers and microservices.

2.2.1 Operating System Containers

"A container is an object isolating some resources of the host, for the application or system running in it." [LXC] A container is, in essence, a very light weight virtual machine whose resources (memory, disk and network) can be very flexibly shared with a host computer or other containers. A container provides some of the isolation properties of an independent computer or a full virtual machine, but a container can be launched in a fraction of a second on commodity hardware. A container will generally require significantly fewer computing and storage resources than a full virtual machine.

Container-based isolation can clearly reduce the impact of software vulnerabilities if the isolation is strong enough. It is therefore important to gain assurance that container infrastructure components (e.g., control groups and name spaces in the Linux kernel) are robust in the face of malicious inputs. Users of containers need to model, analyze, test and validate these to build assurance that container configurations operate as intended.

Although containers can be very lightweight and flexible, the flexibility comes at a cost of complexity in container configurations, which determine numerous critical elements of a container, such as, how it shares its resources, how its network stack is configured, its initial process, the system calls it can use and more. Although the market has already embraced management systems, such as Docker [Docker16], that support the sharing of container configurations, there is a need for tools and techniques that can analyze container configurations and determine the extent to which they reduce security risk, including the extent to which they can mitigate the effects of software vulnerabilities.

Additionally, containers offer an opportunity to apply some of the traditional security models and intrusion tolerance techniques using building blocks that favor efficiency and ease of deployment. There is now a new opportunity to reevaluate which advanced security models and intrusion tolerance techniques can become mainstream technologies.

Furthermore, because a container can be efficiently wrapped around a single run of a program, a container might be configured to grant a program only the minimum level of access to resources, thus following the principle of least privilege [Saltzer75]. Least privilege is a fundamental principle for limiting the effects of software vulnerabilities and attacks. It is notoriously difficult, however, to specify the minimum resources that a program requires. Rather than trying to solve the problem in its full generality, one strategy is to develop analysis techniques and tools to generate custom container configurations that approximate least-privilege for important classes of programs. Due to the relative ease of deploying containers, such tool-assisted containers could bring much more effective access control and safety to mainstream systems.

2.2.2 Microservices

Microservices describe "An approach to designing software as a suite of small services, each running in its own process and communicating with lightweight mechanisms." [Fowler14] The essential microservices idea is not new: it has been explored using web services and in operating systems based on microkernels such as the Mach microkernel [Rashid86], the GNU Hurd [Hurd16] and the Web Services Architecture [WSA04]. The microservices approach, however, structures services according to different criteria. Microservices should implement individual business (or mission) capabilities, have independent refresh cycles, be relatively easy to replace and be programming-language agnostic[Fowler14]. In short, each microservice should make economic and management sense on its own. At the same time, microservices may rely on one another, which can support well-defined modularity.

This approach to system structure can result in a number of components whose interfaces are explicitly defined and whose dependencies are similarly explicitly defined.

As a system operates and the flow of control passes between microservices, there is a natural incentive to “batch up” inter-service communications to amortize boundary-crossing overheads. While this batching can increase latencies in some cases, it can also simplify inter-component dependencies and possibly reduce the likelihood of software flaws and, hence, vulnerabilities.

The deployment of software as collections of microservices raises a fundamental question: does it make sense to build a “trusted microservice”? Even more ambitiously, would it be feasible to develop microservices that are themselves reference monitors? The reference monitor concept dates from the 1972 Anderson Report [Anderson72] and refers to a system component that mediates all accesses to resources that it provides. A reference monitor is: 1) always invoked, 2) tamperproof and 3) verified (i.e., small enough to be built with high assurance). As microservices are becoming increasingly popular, the time may be right to research criteria for formulating microservices that are trustworthy, or that are reference monitors, and to understand the security limitations of the microservices’ architectural patterns.

By making component dependencies and interactions more explicit, microservices appear to offer a new opportunity for interposition-based security enhancements. Wrapping layers inserted between microservice interactions would have the power to augment, transform, deny and monitor those interactions. Those powers could be used to restrict potential damage from software vulnerabilities, but interposition can also destabilize systems and impose slowdowns. A possible research thrust is to investigate interposition strategies that are compatible with microservice based systems. Microservices encourage a simplified interface between components (e.g., no memory sharing via global state). This simplification may enable microservice-based interposition to be performed with fewer associated effects on stability.

2.2.3 Maturity Level

Virtualization systems date from the 1960s. The LXC container form of virtualization began in 2008 and has been under active development since. A number of alternate lightweight virtualization systems exist, for example BSD Jails, OpenVZ and Oracle Solaris Zones. Containers are substantially deployed in clouds and on servers.[1]

The current microservices terminology and design goals emerged by 2014. Earlier formulations, such as tasks running on microkernels, predate the CMU Mach project's initiation in 1985. Since then, microkernel technology has been a subject of ongoing research and has been integrated into significant commercial products, notably Apple's OS X.

2.2.4 Basis for Confidence

The base technologies are widely used, and there is a recognized need for more automation in the configuration of containers. So there could be demand pull. Because containers can be very quickly created, tested and deleted, there is a good case that extensive testing could be done on container configurations in a semi-automated manner. With respect to microservices, growing number of microservice frameworks indicates that the technology is increasing in popularity and that there is still room for enriching microservice frameworks and for having these enrichments adopted. Also, the modular nature of microservices may offer a pathway for deploying more secure versions of microservices without significantly disrupting service to clients.

2.2.5 Rationale for Potential Impact

Operating system containers and microservices are already a significant part of the national information infrastructure. Given the clear manageability, cost and performance advantages of using them, it is reasonable to expect their use to continue to expand. Security-enhanced versions of these technologies, if adopted, can therefore have a widespread effect on the exploitation of software vulnerabilities.

2.2.6 Further Reading

[Fowler14] Martin Fowler, "Microservices: a definition of this new architectural term," 25 March 2014. Available: http://martinfowler.com/articles/microservices.html Accessed 13 October 2016.

[Lemon13] Lemon, "Getting Started with LXC on an Ubuntu 13.04 VPS," 6 August 2013. Available: https://www.digitalocean.com/community/tutorials/getting-started-with-lxc-on-an-ubuntu-13-04-vps Accessed 13 October 2016.

[What] "What's LXC?", Available: https://linuxcontainers.org/lxc/introduction Accessed 13 October 2016.

[1] Operating system containers are useful implementing mechanisms for some clouds, but cloud computing is distinct from operating system containers [Mell11].

2.3 Additive Software Analysis Techniques

Currently there are many different tools and techniques, both as open source and as proprietary software, to analyze software and to check for myriad problems. Many of them can be executed through a general Integrated Development Environment (IDE), such as Eclipse. But current tools face a number of impediments. IDEs sometimes do not offer an "information bus" for tools to share software properties. Each tool must do its own parsing, build its own abstract syntax tree (AST), list variables with their scopes and attributes and "decorate" an AST with proven facts or invariants. Some tools are built on a common infrastructure, such as LLVM or ROSE [Rose16], so they share code, but they must still do much of the analysis over again. In addition, there are few standards that allow, for example, one parser to be swapped out for a new parser that runs faster.

Additive software analysis refers to a comprehensive approach for addressing impediments to the use of multiple advanced software checking tools. The goal of additive software analysis is to foster a continuing accumulation of highly-usable analysis modules that add together over time to continually improve the state of the practice in deployed software analysis. Additive Software Analysis has three parts. First, it is documentary standards to allow algorithms and tools to exchange information about software. Second, it is a framework or architecture to enable modular and distributed development of software assurance and assessment tools. This framework has a function similar to the Knowledge Discovery Metamodel (KDM) [KDM15] or what is termed a black board in Artificial Intelligence (AI). Third, it is conceptual approaches to aggregate, correlate or synthesize the results and capabilities of tools and algorithms. A key output of additive software analysis will be a new generation of user-facing tools to readily combine the outputs from different tools and techniques into unified, more comprehensive assessments of a piece of software.

A comprehensive additive software analysis capability must facilitate tools working together, provide building blocks to jumpstart new tool development and facilitate integration and interoperability among tools. Hence, it must include standards, a framework and techniques to combine analysis results.

2.3.1 Software Information Expression and Exchange Standards

Software assurance tools derive and store an enormous variety of information about programs. Unfortunately, there is no widely-accepted standard for exact definitions of the information or how it might be stored. Because of the lack of standards, developers must perform heroic feats to exchange information with fidelity between different analysis tools and algorithms.

Merely passing bits back and forth between tools is of little benefit unless those bits convey information that is understood the same way by tools. For example, "error," "fault," "failure," "weakness," "bug" and "vulnerability" are related, but different, concepts. Without standards, if one tool reports a bug, another tool may understand "bug" to indicate a higher (or lower!) potential for successful attack than the first tool's assessment.

For example, a variety of formally defined information may be relevant for analyzing a program:

- Location in code;
- The variables that are visible at a certain location, with the variable types;
- Possible values of variables at a certain location. This may include relations between the values of variables, such as x < y;
- Call traces and paths, that is, all possible ways to reach this point;
- Attribution to source code locations for chunks of binaries and executables;
- Possible weaknesses, e.g., possible BOF [Bojanova16], or the input that will be used in an SQL query not filtered and therefore tainted;
- Assertions, weakest preconditions, aspects, invariants and so forth and
- Function signatures, including parameter types.

Program analysis can be applied at various stages of software development and to representations of a program at different levels of abstraction. For instance, tools may operate on the static structure of a program, such as its AST, on representations that represent data or control flow and even on semantic representations that encode functional behaviors, such as weakest preconditions. We look at each of these categories in turn below.

Abstract Representation Early static checkers usually had to include their own parsers for building an AST to analyze. However, compiler writers realized the importance of developing common intermediate representations (IRs) that are well-documented and easily accessible. For instance, in version 4.0, the development team of the GNU compiler, gcc, [GCC16] introduced the intermediate language GENERIC, which is a language-independent format for representing source programs in any of several languages. As another example, the Clang compiler [Clang] provides a well-documented AST that may be either directly accessed by third-party plugins or saved in a common format, such as JSON, to be processed by third-party analysis tools. Other compilers that provide well-documented interchange formats include Frama-C [FramaC] and the ROSE compiler infrastructure [Rose16].

Intermediate Representation Tools may perform in-depth analyses on intermediate representations (IRs) that are closer to the final executable code generated by compilers. For instance, the GNU compiler defines the GIMPLE format in which the original source program is broken down into a simple three-address language. Similarly, the Clang compiler provides the LLVM bitcode representation, a kind of typed assembly language format that is not tied to a specific processor. Others are Common Intermediate Language (CIL) [ECMA12] and the Java Virtual Machine Instruction Set or bytecode [Lindholm15]. The Vine Intermediate Language is a platform-independent machine language [Song08].

Semantic Representations Tools that check functional correctness properties typically need a representation that is more suited to expressing logical program properties than the representations discussed above. While such representations are not as mature as ASTs and compiler IRs, a few have gained popularity in recent years. For instance, the intermediate verification language Boogie [Barnett05], which provides features such as parametric polymorphism, universal and existential quantification, nondeterministic choice and partial orderings, has become a popular backend for sophisticated checkers of both low-level languages, such as C and C++, and higher-level object-oriented languages, such as Eiffel and C#. Boogie programs can be translated into the SMT-LIB format [SMTLIB15], which allows them to be checked with any theorem prover that accepts the SMT-LIB format. Another example of a common language for semantic representations is Datalog [Whaley05], which has been used to build a variety of tools for checking array bound overflows, finding race conditions in multithreaded programs and checking web application security.

2.3.2 Tool Development Framework or Architecture

To foster new tool development, additive software analysis requires initial building blocks. The key initial building block is a framework that can tie the capabilities of tools or techniques together. Just as Eclipse greatly facilitates the improvement of IDE technology for developing code, a framework for additive software analysis will aim to enable synergistic development of software assurance and testing tools. This “framework” may be a separate tool, or it may be a plugin or update to an existing IDE.

Broadly speaking, there are two common methods for frameworks to transmit information between program analysis tools. The first method integrates a checker as a plugin into an existing compiler toolchain. Modern compiler frameworks, e.g., gcc, Clang and Frama-C, make it easy to write new plugins. Furthermore, plugins are often allowed to update an AST or intermediate form, thus allowing plugins to make the results of their analysis available for use by other plugins. For instance, the Frama-C compiler framework provides a library of plugins that includes use-def and pointer-alias analyses that are often necessary for writing semantic analyzers. The second method relies on a common format that is written to disk or sent via network to pass information. An example of this is the Evidential Tool Bus [Rushby05] that allows multiple analysis engines produced by different vendors to exchange logical conclusions in order to perform sophisticated program analyses. Information could be attached to the code to become “assurance-carrying code” [Woody16]. An additive framework would support both information transmission approaches in order to reuse existing efforts as much as possible.

The framework capabilities referred to in this section focus on information exchange among tools, rather than development capabilities of frameworks discussed in Sect. 2.4.

2.3.3 Strategy to Combine Analysis Results

With standards in place and a framework, we can get increased benefit by adding together or combining different software analyses. There are three general ways that results of software analysis can be added together. The first case is simply more information. Suppose the programmer already has a tool to check for injection bugs (INJ class) [Bojanova16]. Adding a tool to check for deadlocks could give the programmer more information.

The second case is confirmatory or contradictory. The programmer may have two different heuristics to find faulty operation (FOP class) bugs [Bojanova16] that have independent chances of reporting true FOP bugs and false positives. The framework could be used to correlate the outputs of the two heuristics to produce a single result with fewer false positives. In contrast, one tool may say that a statement is reachable, while another tool says that it is not. This contradiction may indicate differences in assumptions or an error in a tool. A set of deduction rules used in one tool may be consistent in isolation, but inconsistent with rules from another tool.

The third case of additive software analysis is synergy. A research group with expertise in formal reasoning about memory use and data structures could build upon a component developed by a group that specializes in "parsing" binary code, thus creating a tool that reasons about the memory use of binaries. Developers can experiment with hybrid and concolic assurance tools more quickly. For instance, a tool may use a static analyzer to get the code locations that may have problems then, using constraint satisfiers and symbolic execution, create inputs that trigger a failure at each location.

2.3.4 Technology to Combine Analysis Results

Once trained, such neural networks might serve as vulnerability detection engines in their own right. Incorrect conclusions from manual analysis is both time- and resource-intensive. To be effective, the above strategies to combine results must be instantiated in tools with appropriate technology. For instance, Code Dx [CodeDx15] is a tool that matches, consolidates and presents the output of analysis tools. Even more powerful would be applying machine learning techniques, such as those in Microsoft's Cognitive Services and Computational Network Toolkit and Google's TensorFlow.

The underlying hardware required to perform both the training and inference stages of deep learning and neural networks has dramatically dropped in price. Cloud services now provide high-end graphics processor unit (GPU) instances on demand. This computing capability dramatically increases the speed of machine learning algorithms, allowing for training with much larger datasets and significantly faster training and feedback loops. Organizations that have amassed large corpora of vulnerability, exploit, malware, rootkit and backdoor information can mine this information with well-known data science techniques to derive new insights and value.

Once trained, such neural networks might serve as vulnerability detection engines in their own right. Incorrect conclusions from human practitioners can be fed back into the training phase to automatically strengthen the discrimination of the algorithms or networks and incrementally improve them. These techniques have the potential to detect vulnerabilities across a broad set of languages, including newly launched languages without well-established analysis tools, and even detect new classes of vulnerabilities.

This detection capability could augment existing tool chains. More importantly, results from other analysis tools could be an additional source of information for training and analysis of these powerful techniques. These same networks, given enough resources, can also analyze vast swaths of source code with the same goal of identifying and prioritizing risky components.

2.3.5 Maturity Level

Many commonly used compilers, such as gcc, Clang and Frama-C, provide built-in support for adding plugins that process and update AST and IR representations. Additionally, large communities have developed extensive libraries of plugins and created wiki sites with tutorials and reference manuals that lower the bar for new users to become involved. In the case of semantic representations, the communities are smaller and the bar to entry is higher, although languages like Boogie have been successfully used as the engine by several research groups for building checkers for diverse languages, such as C [VCC13] and Eiffel [Tschannen11], and even an operating system [Yang10].

There are many current software information exchange systems, such as LLVM, ROSE, gcc's GENERIC or GIMPLE and the Knowledge Discovery Metamodel (KDM). Formats to consolidate the output of tools, such as Tool Output Integration Framework (TOIF) and Software Assurance Findings Expression Schema (SAFES) [Barnum12], already implicitly indicate classes of useful knowledge about software.

2.3.6 Basis for Confidence

The leading static analysis tools today have low false positive rates, which has led to increasing adoption throughout industry and government organizations. This in turn has motivated compiler teams to add support for plugins that can operate on internal program representations. There are large and active user communities that are documenting interfaces and creating libraries of plugins that can be combined to build complex analyzers. Indeed, the challenge is not whether an additive software analysis approach might work, but in which to invest and how to tie them together.

2.3.7 Rationale for Potential Impact

Early static analysis tools checked mostly syntactic properties of programs, enforcing coding guidelines and looking for patterns that corresponded to simple runtime errors, such as dereferencing a null pointer or using a variable before assignment. As analyzers became more sophisticated, they increasingly relied on more complex analyses of program structure and data flow. Common frameworks that allow users to build small analysis engines that can share and combine results will make it possible to build sophisticated analyzers. Such analyzers can find subtle errors that are hard to find using traditional testing and simulation techniques.

Such frameworks and standards should allow modular and distributed development and permit existing modules to be replaced by superior ones. They should also facilitate synergy between groups of researchers. They should accelerate the growth of an "ecosystem" for tools and the development of next-generation "hybrid" tools. A hybrid tool might use a static analyzer module to find problematic code locations, and then use a constraint satisfier module and a symbolic execution engine to create inputs that trigger failures. A growing, shared set of problematic and virtuous programming patterns and idioms may ultimately be checked by tools [Kastrinis13].

2.3.8 Further Reading

[Bojanova16] Irena Bojanova, Paul E. Black, Yaacov Yesha and Yan Wu, "The Bugs Framework (BF): A Structured Approach to Express Bugs," 2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2016), Vienna, Austria, 1-3 August 2016, https://doi.org/10.1109/QRS.2016.29.

[Kastrinis13] George Kastrinis and Yannis Smaragdakis, "Hybrid Context-Sensitivity for Points-To Analysis," *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, 2013, pp. 423-434, https://doi.org/2499370.2462191.

[Rushby05] John Rushby, "An Evidential Tool Bus," in *Proc. 7th international conference on Formal Methods and Software Engineering (ICFEM'05)*, Springer, 2005, p. 36, https://doi.org/10.1007/11576280_3.

2.4 More Mature Domain-Specific Software Development Frameworks

Briefly stated, the goal of this approach is to promote the use (and reuse) of well-tested, well-analyzed code, and thus to reduce the incidence of exploitable vulnerabilities.

The idea of reusable software components, organized into component libraries or repositories as mentioned in Sect. 4.3.6, dates from at least 1968 [McIlroy68]. To make software reusable, sharable software components can be packaged in a variety of building blocks, e.g., standalone programs, services, micro-services, modules, plugins, libraries of functions, frameworks, languages (noted in Sect. 2.1.4), classes and macro definitions. A set of such (legacy) building blocks typically forms the starting point for new software development efforts. Or, more colloquially expressed: hardly anything is created from scratch. The vulnerability of new software systems, therefore, depends crucially on the selection and application of the most appropriate existing components and on the interaction of new code with legacy components.

Although the unit of code sharing can be small, e.g., a single function or macro, there are substantial benefits to using mature, high-value, components where significant investments have already been made in design cleanliness, domain knowledge and code quality.

A software framework contains code and, more importantly, also defines a software architecture (including default behavior and flow of control) for programs built using it. A domain-specific framework includes domain knowledge, e.g., GUI building, parsing, Web applications, multimedia and scheduling, as well. A mature domain-specific framework, once learned by software developers, can enable quick production of programs that are well tested both from a software perspective and from a domain knowledge perspective. In the best case scenario, where a mature framework is wielded properly by experts, there is a substantial opportunity to avoid software mistakes that can result in exploitable vulnerabilities.

Unfortunately, the best case scenario is difficult to achieve. Specifically, to realize the benefits of mature frameworks, software developers must overcome several significant challenges.

Finding Suitable Frameworks. A plethora of frameworks exist. For example, a simple search of github.com in September 2016 showed over 171 000 repositories having the word “framework” either in their name or in their description string. (While some of these frameworks reflect significant investments in design cleanliness and quality, others have been hastily built, are of unknown provenance or are possibly malicious.) The frameworks are implemented in a wide variety of programming languages (PHP, JavaScript, Java, Python, C#, C++, etc.), and many frameworks use multiple languages. Additional complexity results from a diversity of package management and build systems that must be learned by potential framework clients. Software development teams confront a significant challenge merely to survey the possible frameworks that might support a project’s requirements; the challenge is acute enough that there is one project [TodoMVC16] that exists solely to help developers choose among available (model-view) frameworks by showing a sample application implemented in multiple frameworks, for comparison purposes. Assessing suitability in surveyed frameworks is a further challenge. Many frameworks include some form of testing in their build processes, often unit testing [Beck94]. Such existing tests need to be assessed for sufficiency relative to a project’s goals. A further issue is that different platforms provide different security features (e.g., access control lists, signed executables and white listing). To the extent that a framework requires a specific platform, choosing that framework must take into account understanding and employing platform-specific assurance features.

Learning New Frameworks. Brooks said that software embodies both “essential” and “accidental” information [Brooks95]. The essential information is about algorithms and fundamental operations that software must perform. The accidental information is about interface details, programming language selection, the names given to elements in a system, etc. Each framework embodies both kinds of information, which must be understood at an expert level to safely employ a framework for nontrivial applications. While an expert might already know much of the essential information for a problem domain, the accidental information cannot be anticipated.

A quick perusal of a common data structure, the list, illustrates the fundamental difficulty. The meaning of a list is well understood by most software developers, but the information required to actually create and use a list data structure is quite different between competing environments. For example, the Unix queue.h macros, Java collections, JavaScript arrays, Python’s built-in list and the C++ Standard Template Library list template, all implement the same basic idea, but using quite different details. A software developer may be an expert in the concept of a list and in some list implementations, but an absolute novice in the usage of the concrete list implementation in a new framework. The developer must, therefore, expend time for the unedifying learning of (often extensive amounts of) accidental information. If developers give in to schedule pressure to minimize this preparatory work, novice-level framework-based software may be produced, which is more likely to contain flaws and vulnerabilities.

Understanding and Controlling Dependencies. One framework may depend on others. The resulting transitive graph of dependencies can be large. Framework users may easily find the vulnerabilities in their projects dependent on possibly voluminous framework code included automatically and indirectly by legacy package managers and build systems. The “leftpad” incident of 2016 illustrates the danger. The heavily-used Node Package Manager maintains numerous packages, which JavaScript programs can easily refer to and use. When an ownership controversy erupted in 2016, an Open Source author unpublished over 250 of his modules from the Node Package Manager. One was the tiny function “leftpad,” which adds padding of spaces or zeros to strings. Thousands of programs, some very important, relied on “leftpad” and suddenly failed until the unpublished package was “un-unpublished” [Williams16].

Resolving Framework Composition Incompatibility. Multiple frameworks may not be usable simultaneously in the same program. Or, if they are, the order of their inclusion or the version may be important, resulting in brittle code. In other cases, such as the lex/yacc code generation tools, explicit actions are needed to avoid name space conflicts in order to allow multiple instances of a framework to coexist in a program. Such conflicts may be subtle. As Lampson points out, each component may have a distinct “world view” and the composition of *n* components can result in n^2 interactions [Lampson04].

These are long-standing challenges. Moreover, due to the large and growing number of frameworks (of varying provenance and quality) currently available in Open Source via public repositories hosted by repository-management entities, such as GitHub, JIRA, Bitbucket, CollabNet, etc., the difficulty of choosing a suitable framework may be more acute. This widespread use, however, also represents an important opportunity: if even small improvements can be achieved to how frameworks are found, learned, dependency-managed and composed, many software vulnerabilities may be avoided.

A second significant development is the mainstreaming of software development (including framework use) through copy/paste operations using software question/answer sites, such as stackoverflow or stackexchange. Although question/answer-based code reuse can be fast, it also can result in poorly-understood and poorly-integrated solutions. The ability to get answers and sample code for questions posed clearly can benefit developer comprehension, however techniques are needed to avoid generating vulnerabilities when adapting others’ solutions.

Although these are significant challenges, the current state of the art provides opportunities to leverage existing code and skills resources while augmenting them with new techniques and tools.

2.4.1 Rapid Framework Adoption

Framework adoption is clearly impeded by the need to learn great quantities of accidental information. Gabriel defines "habitability" as "the characteristic of source code that enables programmers, coders, bug-fixers and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently." [Gabriel96] Recognizing the challenge of achieving habitability, Gabriel suggests the use of software patterns to help developers quickly understand existing code, as well as to flag the use of negative practices. Although not a panacea, patterns (e.g. [Gamma95]) can help bridge the conceptual gap between framework providers and framework consumers. One approach to facilitating this is to develop a set of patterns that encompass popular domains. An informal survey in September 2016 of the top 10 most popular ("star'd") and most "forked" repositories on GitHub shows significant framework activity around Web application development, Front-end Web development, operating system kernels, cross platform application frameworks, virtual machine management, programming languages and asynchronous http servers. One approach to speeding adoption is to formulate software patterns for some of these domains, with a focus on harmonizing the accidental information between frameworks (so it need not be learned multiple times) and to produce documentation for common use cases. Experiments can then measure the effectiveness by comparing framework uptake both with and without the new pattern information.

2.4.2 Advanced Test Methods

With the unique exception of civil aviation (where software is built and tested according to stringent Federal Aviation Administration (FAA) standards), commercial software often receives only minimal testing. This is particularly true in industries where time to market is the predominant goal. Testing may be limited to only basic assurances that required functions have been implemented, sometimes called "happy path" or positive testing that demonstrates features, but does not provide assurance that undesirable behavior is absent.

Advanced testing approaches hold the promise to substantially increase framework robustness, and furthermore, to build assurance for compositions of frameworks under various assumptions regarding dependencies. Many frameworks currently employ only ad hoc testing. Others employ standard unit testing [Beck94], practiced at varying levels of completeness. Automated testing, such as QuickCheck [Claessen02] or various fuzzing tools, is available for most languages. Recent advances in the measurement of traditional test suite coverage provide an opportunity to compare frameworks. Combinatorial testing compresses huge numbers of combinations of input values into a small number of tests [Kuhn10]. The many ways in which frameworks may be customized or configured suggest a possible approach for gaining new confidence in the use of software frameworks. By demonstrating high quality compositions, such testing also has potential to highlight framework similarities, reduce learning curves and enable broader adoption of well-tested, well-analyzed code.

2.4.3 Conflict Resolution in Multi-Framework Composition

In some cases, multiple frameworks can be used together concurrently without conflict. In others, the composition details that allow concurrent use may be fragile. Dominant framework patterns, such as inversion of control (IoC) [Busoli07]—also known as the Hollywood principle: “don’t call us; we’ll call you,” may exacerbate this, because each framework may assume that it is defining the flow of control in an entire application. One approach for mitigating this is to virtualize framework operations using, for example, lightweight operating system containers [LXC] and then establish communication links between concurrently executing frameworks. Another approach to conflict resolution is to employ software translation to rewrite frameworks, so that their overlapping elements become distinct. Pilot efforts can demonstrate the feasibility of these and other deconfliction strategies and compare their costs and effects on application vulnerability.

2.4.4 Maturity Level

The literature of software patterns is quite extensive and software testing is a relatively mature subfield of computer science, practiced now for over 40 years. Frameworks themselves are now a dominant unit of software sharing. The three techniques listed above, patterns, testing and frameworks, are under continuous use and refinement.

2.4.5 Basis for Confidence

There is little doubt that patterns can be documented for several significant frameworks; rapid uptake may be a more incremental than revolutionary improvement, but incremental improvements should flow from investments in pattern documentation. The advanced testing techniques that would be brought to bear on framework compositions, are relatively mature, increasing confidence that framework integrations can be effectively tested.

2.4.6 Rationale for Potential Impact

Code reuse is pervasive and seemingly accelerating; by investing in very popular frameworks, any improvements will be widely relevant.

2.4.7 Further Reading

[Software16] “Software framework." Available: https://en.wikipedia.org/wiki/Software_framework Accessed 13 October 2016.

[TodoMVC16] “TodoMVC: Helping you select an MV* framework." Available: http://todomvc.com/ Accessed 13 October 2016.

[Wayner15] Peter Wayner, “7 reasons why frameworks are the new programming languages," 30 March 2015. Available: http://www.infoworld.com/article/2902242/application-development/7-reasons-why-frameworks-are-the-new-programming-languages.html Accessed 13 October 2016.

2.5 Moving Target Defenses (MTD) and Automatic Software Diversity

This approach is a collection of techniques to automatically vary software’s detailed structures and properties such that an attacker has much greater difficulty exploiting any weakness. To illustrate, consider one recently-proposed technique in this family: heap memory randomization [Iyer10]. When a program requests a buffer, the easiest thing is to return a chunk of the next available memory. This puts buffers in the same relative location. Knowing this, an attacker can exploit a buffer overflow weakness (i.e., BOF class) [Bojanova16] in one buffer to, for instance, read the password in another buffer that is always 384 bytes beyond it. Heap memory randomization allocates a random additional amount of memory during each allocation. This puts buffers in different (unpredictable) relative locations, so that the above exploit is much harder or possibly impossible.

The goal of software diversity and moving target defense (MTD) is to reduce an attacker's ability to exploit vulnerabilities in software, not to reduce the number of weaknesses in software. This reduction may be achieved by changing the “attack surface,” that is, the interface accessible by the attacker either across time (changes during operation) or across copies (diversity). Reduction may also include regenerating system components that have been compromised [Knight12].

Diversification must, of course, be safe. That is, changes have no effect on normal behavior, other than perhaps higher use of resources. Even with this constraint we can trade computing power for increased granularity or thoroughness of diversification. The increased granularity is presumed to offer better protection against exploitation of unknown vulnerabilities, because of the higher probability of affecting the location or value of some piece of information essential to an attack. This tradeoff is similar to that for static analysis, referred to in Sect. 2.1.1 and 2.1.2: the more resources invested, the higher the amount of assurance. The difference is that static analysis finds specific vulnerabilities so that they can be corrected, while diversity-inducing transformations increase the difficulty of exploiting whole classes of vulnerabilities.

2.5.1 Compile-Time Techniques

Compile-time techniques are those applied automatically by a compiler. They may result in the same executable for each compilation, such that the executable then chooses random behaviors or memory layouts at run time, or they may result in a different executable at each compilation.

Some specific techniques are data structure layout randomization, different orders of parameters in function calls, address space layout randomization (ASLR) [PaX01], instruction set randomization, data value randomization, application keyword tagging and varied instruction ordering with operation obfuscation and refactoring.

The program information that is useful for proving that these diversifications are safe is also useful for program analysis to find or remove vulnerabilities. The additive software analysis approach, detailed in Sect. 2.3, is to use the same computing power to simultaneously detect or remove weaknesses and to also randomize remaining weaknesses. These diversification techniques could be tied into a static analysis tool through the additive analysis framework, potentially with very modest resource expenditures.

Unfortunately, no tools do this today. Analysis software is usually run by the programmer, at development time. Diversification typically only displays its benefit in the system test phase or in the operation phase when it demonstrates resilience. At worst, diversification adds ambiguity to test results and makes it more difficult to track down root causes of failures. To counteract this disconnect between effort and benefit, programs that use diversification should be specifically acknowledged, so customers know that they employ an extra layer of resilience. Compilers that produced extremely diverse results would also reduce the chance of adversaries discovering vulnerabilities by examining differences between the new and previous versions of a program: they would be too different [Franz10].

2.5.2 System or Network Techniques

Some techniques at the system or network level are network address space randomization and protocol diversity [Rowe12]. These are likely to be dynamic in that they change on a regular basis. In many cases, these are built on the assumption of a shared secret map from services to addresses or a shared secret key, so an application can authenticate and get current information.

2.5.3 Operating System Interface Techniques

An operating system (OS) may present different interfaces to different processes. These could be dynamic, such as a random interrupt number assigned for each system service, or static, in which the OS has several choices for each set of services. In the dynamic case, the linker/loader can adjust each new executable to the assignments made for the process. As an example of the static case, an OS presents a new process with a set of j memory management APIs, a set of k process services, a set of m networking functions and a set of n I/O calls. Invasive code trying to execute through that process would have to deal with $j \times k \times m \times n$ different OS interfaces to succeed.

2.5.4 Maturity Level

Some moving target defenses are the default in many operating systems and compilers today. There are intense research and entire conferences focused on understanding limitations, costs and benefits of current techniques and developing new and better techniques.

2.5.5 Basis for Confidence

The benefit in terms of number of attacks foiled, attackers discouraged or additional attacker resources required is not known. However, many MTD techniques can be applied automatically, e.g. by the compiler, at little cost of resources or run time.

2.5.6 Rationale for Potential Impact

MTD techniques can be applied to most programs and systems today, even static embedded systems. Thus, the scope of benefits is extremely large. The impact is not clear since most techniques increase attacker's costs, not strictly eliminate vulnerabilities.

2.5.7 Further Reading

[Okhravi13] H. Okhravi, M. A. Rabe, T. J. Mayberry, W. G. Leonard, T. R. Hobson, D. Bigelow and W. W. Streilein, "Survey of Cyber Moving Targets," Massachusetts Institute of Technology Lincoln Laboratory. Technical Report 1166, 25 September 2013. Available: https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full_papers/2013_09_23_OkhraviH_TR_FP.pdf Accessed 13 October 2016.

3 Measures and Metrics

This section deals with measures, assessments, metrics, appraisals, judgements, evaluations, etc. in the broadest sense. Hence, code reviews, software testing and other techniques have a place in this section. As we discuss later, there is a dearth of precisely defined, rigorously validated measures. Worse, most existing measures are only moderately predictive of the high-level properties that we wish to determine in software. There is not even extensive and detailed data, such as numbers and types of vulnerabilities, upon which measurement research might be based.

We have three areas of attention. First, encouraging the use of measures. All the extraordinary measures in the world do not help if nobody uses them. Also, nobody *can* act on measures if the measures are not produced and available. The Federal Government might motivate and encourage the use of software product measures. Vehicles include procurement, contracting, liability, insurance and also standards, as explained in Sect. 4.3. The benefit of measures is amplified when they are revised, interpreted and used in a disciplined software development process [Curtis16]. Indeed, the widespread use of good measures is one of the few ways with the potential to break out of the cycle of crash-and-patch and get ahead of attackers [Grigg08]. Software can also benefit from the programs and criteria of third-party, non-governmental organizations. Some possibilities are UL Cybersecurity Assurance Program (CAP), Consortium for IT Software Quality (CISQ) Code Quality Standards, Coverity Scan, Core Infrastructure Initiative (CII) Best Practices badge and the Building Security In Maturity Model (BSIMM). Many of these include process measures, which is the second area of attention.

The second area, process measures, includes hours of effort, number of changes with no acceptance test defects and acceptance test defect density in delivered code [Perini16]. These do not have a direct effect on the number of vulnerabilities, but the indirect effects are significant. For example, if developers are forced to frequently work overtime to meet a deadline or the schedule does not allow for training, the number of vulnerabilities is likely to be much higher [Perini16]. Other examples are measures that indicate how much a new process step helps compared to the former practice or that indicate parts of the process that are allowing vulnerabilities to escape. This approach of continuously improving the process is found in the highest levels of maturity models. It also allows groups to adopt or adapt methods and measures that are most applicable to their circumstance. We do not discuss process measures further.

The final area of attention is measures of software as a product, for instance, proof of absence of buffer overflows, number of defects per thousand lines of code, assurance that specifications are met and path coverage achieved by a test suite. The Software Quality Group at the U.S. National Institute of Standards and Technology (NIST) organized a workshop on Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV) to gather ideas on how the Federal Government can best identify, improve, package, deliver or boost the use of software measures to significantly reduce vulnerabilities.[2] They called for short position statements, and then invited workshop presentations based on 10 of the 20 statements submitted. The workshop was held on 12 July 2016. The full workshop report is available as NIST SP 500-320 [Black16]. Much of this section is informed by the results of the workshop. Ideas were often brought up by one person, discussed and elaborated by others, then written or reported by yet others. Hence, it is difficult to attribute ideas to particular people in most cases. We thank all those who participated in the workshop and made contributions, large and small, to the ideas noted in the report.

We distinguish between base measures and derived measures. A base measure is a simple, basic assessment or count with a clear value. A derived measure, on the other hand, is "a function of two or more values of base measures" [ISO15939] or a mathematical transformation of a base measure [ISO25040]. Derived measures are often surrogates for properties that we would like to be able to determine. For instance, number of buffer overflow (BOF class) [Bojanova16] weaknesses is a base measure with a reasonably clear definition. In contrast, code security is a derived measure that is only weakly predicted by the number of BOFs. The absence of flaws does not indicate the presence of excellence.

3.1 A Taxonomy of Software Measures

Software measures may be classified along four dimensions. The first dimension is how "high-level" is the measure. Low-level measures are below semantics, such as size of a program, number of paths and function fan in/fan out. High-level measures deal more with what the program is meant to accomplish. The second dimension is static or dynamic. Static measures are those that apply to the source code or "binary" itself. Dynamic measures apply to the execution of the program. The third dimension is the point of view. It may be either an exterior view, sometimes called black box or functional, or an interior view in which the code is considered, referred to as white ("clear" or "transparent") box or structural. The fourth dimension is the object of the measure: bugs, code quality and conformance.

Along the first dimension, measures may be divided into whether they are low level or high level. Low-level measures are widely applicable. High-level measures, in contrast, deal with the relation between the program, as an object, and the developer or user, as a sentient subject. Quality arises in this interaction between object and subject [Pirsig74]. Analogously to low- and high-level measures, there are low-level vulnerabilities and there are high-level vulnerabilities. Some low-level vulnerabilities are buffer overflow, integer overflow and failure to supply default switch cases. These low-level vulnerabilities can be discerned directly from the code. That is, one can inspect the code or have a program inspect the code and decide whether there is a possibility of a BOF given particular inputs. There is no need to refer to a specification, requirement or security policy to determine whether a buffer overflow is possible.

[2] The web site is https://samate.nist.gov/SwMM-RSV2016.html.

On the other hand, high-level vulnerabilities cannot be discerned solely by reference to the code. A human reviewer or a static analyzer must consult requirements, specifications or a policy to determine high-level problems. For instance, failure to encrypt sensitive information generally cannot be discerned solely by code inspection. Of course, heuristics are possible. For example, if there is a variable named "password," it is reasonable for a static analyzer to guess that variable is a password and should not be transmitted without protection or be available to unauthorized users. But neither tool nor human can determine whether or not the information in a variable named "ID" should be encrypted or not without examining an external definition.

Having access to a requirements document for a security policy does not allow the quality of software to be assessed in all cases. Requirements documents typically deal with the behavior of the program and what the program uniquely needs to do. It is difficult, and perhaps impossible, to specify formally that code should be high quality. Software architecture is an approach to define the structural components that distinguish good and useful software from software that is error-prone, difficult to debug, brittle or inflexible.

The second dimension of classifying measures—static or dynamic—is most apparent in testing. Test measures conceptually have two parts: test generation or selection and test result evaluation. Test measures generally answer the question, how much of the program (interior) or the input space (exterior) has been exercised? Test case generation is necessarily static, while evaluation is usually dynamic, that is, based on the result of executions. In many test measures, the two parts are tied to each other. They may include a step like choose additional test cases to increase the coverage. Thus, the dynamic part influences the static part. Testing is usually referred to as a dynamic technology, since program execution is an essential part of testing. That is, if one comes up with test cases *but never runs them*, then no assurance is gained, strictly speaking. Of course, in most cases the thought and scrutiny that goes into selecting test cases is a static analysis that yields some assurance about the program. ISO/IEC 25023 refers to static measures as internal measures and dynamic measures as external [ISO25023].

The third dimension is the point of view, either exterior or interior. Exterior measures are typically behavioral conformance to specifications, requirements or constraints. They are based on what the software is observed to do. They are often referred to as "black box" or behavioral. These measures are particularly useful for acceptance testing and estimating user or mission satisfaction. It matters little how well the program is written or is structured internally if it does not fulfill its purpose. In contrast, interior or structural measures primarily deal with, or are informed by, the code's architecture, implementation and fine-grained operation. Interior measures are based on analysis of the source code or executable. Measures in this class are related to qualities, such as maintainability, portability, elegance and potential. For instance, exterior timing tests may be insufficient to determine the order of complexity of an algorithm, whereas code examination may clearly show that the algorithm is order $\Theta(n^2)$ and will have performance issues for large inputs.

Determining how much testing is enough also shows the difference between interior and exterior measures. Exterior measures, such as boundary value analysis [Beizer90] and combinatorial testing [Kuhn10], consider the behavior or specification to compute how much has been tested or what has not been tested. On the other hand, interior measures include counts of the number of blocks, mutation adequacy [Okun04] and path coverage measures [Zhu97]. The two approaches are complementary. Exterior-based testing can find missing features. Interior-based testing can bring up cases that are not evident from the requirements, for example, switching from an insertion sort to a quick sort when there are many items.

The fourth dimension to classify measures conceptually divides them into three types. The first type of measure is presence (or absence) of particular weaknesses, such as buffer overflow (BOF class) or injection (INJ class) [Bojanova16]. Note that the absence of flaws does not indicate, for instance, resilient architecture. The second type is quality measures meant to determine the excellence of code, or parts of it. However, we only have proxies for “quality,” for example maintainability, portability and the presence of assertions. Even many of these proxies can only be estimated indirectly. These first two types are product quality characteristics [ISO25010]. The third type is conformance to specification or correctness. This third type of measure is for quality in use characteristics [ISO25010] and must be specific to each task. General requirement languages and checking approaches are available. Because of the profound differences between these three types, there is no one security or vulnerability measure that guarantees excellent code.

3.2 Software Assurance: The Object of Software Measures

Assurance that software will function as it should, comes from three sources. The first is the development process. If software is developed by a team who has clear requirements, are well trained and have demonstrated building good software with low vulnerability rates, then we have confidence or assurance that software they produce is likely to have few vulnerabilities. The second source of assurance is our analysis of the software. For instance, code reviews, acceptance tests and static analysis can assure us that vulnerabilities are likely to be rare in the software. We can trade off these two sources of assurance. If we have little information about the development process or the development process has not yielded good software in the past, we must do much more analysis and testing to achieve confidence in the quality of the software. In contrast, if we have confidence in the development team and the development process, we only need to do minimal analysis in order to be sure that the software follows past experience.

The third source of software assurance is a resilient execution environment. If we do not have confidence in the quality of the software, then we can run it in a container, as explained in Sect. 2.2.1, give it few system privileges and have other programs monitor the execution. If any vulnerabilities are triggered, the damage to the system is controlled.

We may express our assurance with a mathematical formula: $A = f(p, s, e)$, where A is the amount of assurance we have, p is the assurance that comes from our knowledge of the process, s is assurance from static and dynamic analysis and e is the assurance that we gain from strict execution environments. The combining function, *f*, is some additive operation, akin to $p + s + e$. As we said before, if we do not have information about the process, then extra work in analysis can raise our overall assurance. On the other hand, if we have great confidence in the process (and people) that developed the software, we do not need as much work in analysis.

3.3 Software Metrology

To have a coherent, broadly useful system of measures, one must have a solid theoretical foundation, that is, a philosophy of software measurement. This philosophy must have a solid mathematical foundation, for instance, to use sensible statistics [Böhme08]. This section addresses questions such as, what is software metrology? What is its purpose? What are the challenges unique to measuring software, in contrast to physical measurement? What are possible solutions or potential approaches?

Software measures have well-known theoretical limitations. Analogous to Heisenberg’s Uncertainty Principle in Physics, Computer Science has the Halting Problem, Rice’s Theorem and related results that show that it is impossible to correctly determine interesting measures for *all* possible programs. Although this is a caution, it does not mean that all useful, precise, accurate measurement is impossible. There are several ways to avoid these theoretical road blocks. First, we may be satisfied with relative properties. It may be helpful to be able to determine that the new version of a program is more secure (or less!) than the previous version. We need not have an absolute measure of the security of a program. Second, a measure might apply only to programs that have “reasonable” structures. A measure may still be useful even if it does not apply to programs consisting solely of millions of conditional go-to statements with seemingly capricious computations interspersed. Nobody (should) write programs like that. Finally, society may decide that for certain applications, we will only build measurable software. Civil architects are not allowed to design buildings with arbitrary arches, domes, cantilevers and facades. They are required to run analyses showing that the design withstands expected loads and forces before construction can start. Currently most programmers learn to write “elegant” software, then try to show that it works. The expectation might change so that professionals only write software that definitely satisfies its constraints and requirements.

Computer programmers use the phrase “it’s not a bug: it’s a feature” half-seriously. Its use highlights that bugs and features are entities that are related somehow. Let us assume that a program can be characterized as a set of features. (The notion that a program is a set of features is the basis of some size measures. For example, the Function Point measure attempts to capture the notion of a basic operation or function.) Saying that a program “has a bug” means it is a buggy version of a “good” program. Both the good program and the buggy version are programs. According to the assumption, both programs are a set of features. Therefore, the difference between the good program and the buggy program is some set of features — features added, removed, or changed. Hence, a precise definition is that a bug is the difference between the features desired and the features present. In many cases, a bug may merely be an additional feature or one feature replacing another.

We might contrast software metrology with physical metrology. In physical metrology the challenge is to precisely and reproducibly determine the properties of physical objects, events or systems. For software, on the other hand, most of the so-called measurement is merely counting. A case in point is that ASCMM-MNT-7: Inter-Module Dependency Cycles has a precise definition [OMG16]. It is not difficult to write a program that precisely measures the number of instances where a module has references that cycle back a piece of software. The difference then is that physical metrology has clearly identified the properties that they want to determine, for instance, mass, length, duration and temperature. In contrast, software metrology has a distinct gap. We want to measure high-level properties, such as quality, maintainability and security, but we do not have precise definitions of those. Therefore, we cannot measure them directly. We can, however, measure many properties that are correlated with those high-level properties.

Currently metrology relegates counting the number of entities to a second-class method of determining properties. Such counted *quantities* are all considered to be the same dimension one, sometimes called dimensionless quantities, although they may be different *kinds*.

3.4 Product Measures

As much as good process is essential to the production of code with few vulnerabilities, the ultimate capability is to measure the code itself. As pointed out in the introduction to this section, measures of the software itself inform process improvement.

Security or vulnerability measurement in the broadest sense includes both testing and checking. We need such measurements to determine whether the goals of this report are met! Adopting any one technique given in this report may not reduce vulnerabilities dramatically. The lack of reduction may merely be the result of certain details of how the technique is employed, or it may be that the technique is just not applicable. When several techniques are adopted, it is even harder to distinguish the effects of each or joint effects.

Such measurement cannot be left until the final deployment. They must be included in *all* phases of software development. Except for ambitious approaches like the Clean Room approach [Mills87], this kind of measurement cannot be left as a gate near the end of the production cycle.

It is possible that software quality and security measures may be the wrong emphasis to reduce software vulnerabilities. Such measures may fade in emphasis as other software measures have, for example cohesion and McCabe Cyclomatic Complexity. It may turn out that the best approach is like Clean Room, in which measures inform a decision to accept or reject and do not purport to establish an absolute certification of freedom from errors.

3.4.1 Existing Measures

There are hundreds of proposed software measures, such as lines of code, class coupling, number of closed classes, function points, change density and cohesion. Most of these are not precisely defined and are not rigorously validated. Worse yet, most of these are only moderately predictive of the high-level properties that we wish to determine in software. For instance, lines of code (LoC) capture only some of the variance in program capability. LoC for the same specification in the same language varies by as much as a factor of four, even when all programmers have similar expertise. On the other hand, LoC has a remarkably robust correlation with the number of bugs in a program. (This suggests that higher level and domain-specific languages, which allow a programmer to express functionality more succinctly, lead to fewer bugs in general.)

Even something as seemingly simple as counting the number of bugs in a program is surprisingly complicated [Black11b]. It is difficult to even subjectively define what is a bug. For example, one can write a binary search that is never subject to integer overflow, but the code is hard to understand. Dividing by zero may have a well-defined behavior, resulting in the special value “NaN,” but that is generally not a useful result. Bugs are often a cascade of several difficulties. Suppose (1) an unchecked user input leads to (2) an integer overflow that leads to (3) a buffer being allocated that is too small that causes (4) a buffer overflow that finally leads to (5) information exposure. Do we count this as one bug or five? If a programmer makes a systematic mistake in several places, e.g., not releasing a resource after use, is that one problem or several? Rather than being the exception, these kinds of complications are the rule in software [Okun08].

For any realistic program, it is infeasible to test every single possible input. Instead, one must choose a measure that spans the entire space. Some of these measures are combinatorial coverage of input space [Kuhn13], mutation adequacy [Okun04], path coverage measures [Zhu97] and boundary value analysis [Beizer90]. These measures are interrelated. For instance, certain levels of (static) combinatorial coverage produce tests that yield complete branch coverage, a dynamic measure [Kuhn15].

There are far too many proposed measures to evaluate or even list here. We can state that, as alluded to above, measures should be firmly based on well-established science and have a rational foundation in metrology to have the greatest utility [Flater16].

3.4.2 Better Code

Two presentations at the Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV) workshop, Andrew Walenstein’s “Measuring Software Analyzability” and James Kupsch’s “Dealing with Code that is Opaque to Static Analysis,” point the direction to new software measures. Both stressed that code should be amenable to automatic analysis. Both presented approaches to define what it means that code is readily analyzed, why analyzability contributes to reduced vulnerabilities and how analyzability could be measured and increased.

There are subsets of programming languages that are designed to be analyzable, such as SPARK, or to be less error-prone, such as Less Hatton's SaferC. Workshop participants generally favored using better languages, for example, functional languages, such as F# or ML. However, there was no particular suggestion of *the* language, or languages, of the future.

We note that with few exceptions, such as Ada 2012 [Barnes13], which has SPARK, new languages have poor tool support. Supporting the construction of tools is vital for the adoption and safe use of new languages.

While code-based measures are important, we can expect complementary results from measures for other aspects of software. Some aspects are the software architecture and design erosion measures, linguistic aspects of the code, developers' backgrounds and measures related to the software requirements.

3.4.3 Measures of Binaries and Executables

Some workshop participants were of the opinion that there is a significant need for measures of binaries or executables. With today's optimizing compilers and with the dependence on many libraries delivered in binary, solely examining source code leaves many avenues for appearance of all sorts of vulnerabilities.

3.4.4 More Useful Tool Outputs

There are many powerful and useful software assurance tools available today. No single tool meets all needs. Accordingly, users should use several tools. This is difficult because tools have different output formats and use different terms and classes. Tool outputs should be standardized. That is, the more there is common nomenclature, presentation and detail, the more feasible it is for users to combine tool results with other software assurance information and to choose a combination of tools that is most beneficial for them.

In addition, tools can supply more information about their analysis. Tools could indicate which parts of code are thoroughly checked and which parts are not, for instance, because of complexity or heuristics. This checking information could be attached to the code as "assurance-carrying code" [Woody16], analogous to proof-carrying code.

Participants felt the need for scientifically valid research about tool strengths and limitations, mechanisms to allow publication of third party evaluation of tools, a common forum to share insights about tools and, perhaps, even a list of verified or certified tools.

3.5 Further Reading

[Barritt16] Keith Barritt, "3 Lessons: FDA/FTC Enforcement Against Mobile Medical Apps," 14 January 2016. Available: http://www.meddeviceonline.com/doc/lessons-fda-ftc-enforcement/against-mobile-medical-apps-0001 Accessed 12 October 2016.

[FTC16] Federal Trade Commission, “Mobile Health App Developers: FTC Best Practices,” April 2016. Available: http://www.ftc.gov/tips-advice/business-center/guidance/mobile-health-app-developers-ftc-best-practices Accessed 13 October 2016.

[Perini16] Barti Perini, Stephen Shook and Girish Seshagiri, “Reducing Software Vulnerabilities – The Number One Goal for Every Software Development Organization, Team, and Individual,” ISHIPI Technical Report, 22 July 2016.

4 Non-Technical Approaches and Summary

In response to the February 2016 Federal Cybersecurity Research and Development Strategic Plan, OSTP asked NIST to identify ways to dramatically reduce software vulnerabilities. NIST worked with the software assurance community to identify five promising approaches. This report presents some background for each of the approaches along with a summary statement of the maturity of the approach and the rationale for why it might make a dramatic difference. Further reading was provided for each approach. Hopefully other approaches will be identified in the future.

These approaches are focused on technical activities with a three to seven-year horizon. Many critical aspects of improving software, such as creating better specifications, using the testing tools available today, understanding and controlling dependencies and creating and following project guidelines, were not addressed. While these areas fall outside the scope of the report, they are critical both now and in the future. Similarly, the report does not address research and development that is needed as part of a broader understanding of software and vulnerabilities. Topics such as identifying sources of vulnerabilities, how vulnerabilities manifest as bugs and improved scanning during development are also critical, but, again, outside the scope of this report.

This section of the report outlines some of the needed steps for moving forward by engaging the broader community, including researchers, funders, developers, managers and customers/users. This section addresses: 1) engaging and supporting the research community, 2) education and training and 3) empowering customers and users of software to meaningfully participate by not only asking for quality, but pushing it.

4.1 Engaging the Research Community

There are many approaches to engaging the research community beyond simply funding secure software research.

4.1.1 Grand Challenges, Prizes and Awards

Many organizations have announced grand challenges, some of which are general research goals and some are competitions. More secure software can be the focus of challenges or a side benefit, that is, the competition could be focused on a non-security goal, but require the winner to produce secure software. For example, the DARPA Cyber Grand Challenge scores reflected how well software found vulnerabilities and protected the host [DARPA16]. Other challenges might focus on particular techniques, such as abstract interpretation or symbolic execution or analysis of new programming languages. Many organizations use bug bounty programs to provide incentives to the research community to find and notify organizations about bugs.

4.1.2 Research Infrastructure

There are several very successful repositories of data related to secure software, such as the National Vulnerability Database. However, many more are needed. There could be repositories to share related research as well as open repositories of source code, as mentioned in Sect. 4.3.6. There is also a need for a better understanding of weaknesses and bugs. For example, what proportion of vulnerabilities result from implementation errors and what proportion from design errors? The SPSQ workshop participants called for more in-depth understanding of defects and vulnerabilities. Specific issues included the need for empirical data about the types and prevalence of vulnerabilities, the effectiveness of programming and testing techniques and the benefits and costs of "safer" languages. New languages require new analysis tools and, in some cases, new analysis algorithms. A robust research infrastructure can also be used to study other factors that may affect software quality, including management practices, education and training, levels of complexity and programmer overload. Researchers need to be able to replicate results and test across different types of code. All of these activities require a large and public research infrastructure.

4.2 Education and Training

The role of education and training cannot be overstated. There is no technological substitute for developer discipline. Education is not just about teaching developers how to write better software. It also includes educating users how to specify better software and managers how to set up environments that result in higher quality software.

In addition, education and training are the primary mechanism for transitioning the technical approaches discussed in this report from the research community to both the development community and to the user/customer community.

Education and training for the developer community needs to address both up-and-coming developers currently in the educational system, as well as current developers who need to update their skills.

Over the past couple of years, there has been a shift in focus in higher education to include a greater emphasis on designing software with security built in from the beginning rather than added afterwards. K-12 education has also seen growth in cybersecurity efforts – both from the user and producer perspectives. It is clear that computer science and cybersecurity come together in the issue of secure programming. Understanding the principles of cybersecurity are essential to making sure that software is secure and usable. More and more academic programs are educating their students to program with security in mind.

Current developers need to be exposed to new approaches and techniques. In order for developers to make changes, they need to see evidence that the new approaches and techniques will be effective, as well as training material. To complement the training of front-line software developers, managers and executives must also be educated in the risk management implications of software vulnerabilities and the importance of investing in cybersecurity and low vulnerability software. In order for this training to be successful, it, too, will require evidence that investment in secure software will be cost effective.

It is currently unknown which pedagogical techniques are most effective. Early research has shown that providing developers with a better understanding of weaknesses creates better programs [Wu11]. Additional research, as well as training material ranging from use cases to how-to guides, will be needed for successful transition. The Federal Government can lead by example by training its developer community.

The National Initiative for Cybersecurity Education (NICE)'s strategic plan [Plan16] lays out three goals for improving education and training.

Accelerate Learning and Skills Development. It is critical to inspire a sense of urgency in both the public and private sectors to address the shortage of skilled cybersecurity workers. Needed steps include:

- Stimulating the development of approaches and techniques that can more rapidly increase the supply of qualified cybersecurity workers;
- Advancing programs that reduce the time and cost for obtaining knowledge, skills and abilities for in-demand work roles;
- Engaging displaced workers or underemployed individuals who are available and motivated to assume cybersecurity work roles;
- Experimenting with the use of apprenticeships and cooperative education programs to provide an immediate workforce that can earn a salary while they learn the necessary skills and
- Exploring methods to identify gaps in cybersecurity skills and raise awareness of training that addresses identified workforce needs.

Nurture a Diverse Learning Community. There is a need to strengthen education and training across the ecosystem to emphasize learning, measure outcomes and diversify the cybersecurity workforce. Needed steps include:

- Improving education programs, co-curricular experiences and training and certifications;
- Encouraging tools and techniques that effectively measure and validate individual aptitude, knowledge, skills and abilities;
- Inspiring cybersecurity career awareness with students in elementary school, stimulate cybersecurity career exploration in middle school and enable cybersecurity career preparedness in high school;

- Growing creative and effective efforts to increase the number of women, minorities, veterans, persons with disabilities and other underrepresented populations in the cybersecurity workforce and
- Facilitating the development and dissemination of academic pathways for cybersecurity careers.

Guide Career Development and Workforce Planning. Employers need help to address market demands and enhance recruitment, hiring, development and retention of cybersecurity talent. Needed steps include:

- Identifying and analyze data sources that support projecting present and future demand and supply of qualified cybersecurity workers;
- Publishing and raising awareness of the National Cybersecurity Workforce Framework and encourage adoption;
- Facilitating state and regional consortia to identify cybersecurity pathways addressing local workforce needs;
- Promoting tools that assist human resource professionals and hiring managers with recruitment, hiring, development and retention of cybersecurity professionals and
- Collaborating internationally to share best practices in cybersecurity career development and workforce planning.

4.3 Consumer-Enabling Technology Transfer

One of the drivers for better software is if users, consumers and purchasers of software demand it. While the user community clearly wants higher quality software, it is difficult for them to meaningfully ask for it and know if they received it, and thus signal the development of low vulnerability software. The market needs improved measures that are customer-focused, as well as other policy and economic approaches. For a measure to significantly inform customers, it requires pervasiveness, understandability, simplicity and efficiency. An example is the 5-Star Safety Rating of the National Highway Traffic Safety Administration (NHTSA). Once ratings consistently appeared on new automobiles, one- and two-star rated cars rapidly became scarce [Rice08]. Policy and economic approaches are outside the scope of this report, but they are critical to successful technology transfer for improved software. This section outlines some of these approaches that were discussed during the various workshops.

4.3.1 Contracting and Procurement

The Federal Government could lead a significant improvement in software quality by requiring software quality during contracting and procurement and by changing general expectations. Model contract language can include incentives for software to adhere to higher coding and assurance standards or punitive measures for egregious violations of those standards. Sample procurement language for cybersecurity and secure software has been published by the defense community [Marien16], the financial sector, the automotive sector and the medical sector. The evaluation must include provisions for “fitness for purpose” that factor in considerations for secure software.

4.3.2 Liability

There is much discussion in the software community about liability, including during the Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV) workshop. Many participants felt that companies developing software should be contractually liable for vulnerabilities discovered after delivery. Many did not believe that there should be legal liability at this time. On the other hand, the language of such liability clauses needs to be strict enough to, as one participant wrote, "hold companies accountable for sloppy and easily-avoidable errors, flaws and mistakes."

Defining "sloppy and easily avoidable" is not a trivial matter. An additional complicating factor is that liability includes a concept of who is responsible. Responsibility may be hard to determine in the case of "open source" or freely available software.

4.3.3 Insurance

Cyber insurance is a growing area as cyber continues to grow in importance. The Financial Services Sector Coordinating Council (FSSCC) for Critical Infrastructure Protection and Homeland Security produced a 26-page document, entitled "Purchasers' Guide to Cyber Insurance Products," defining what this kind of insurance is, explaining why organizations need it, describing how it can be procured and giving other helpful information.

4.3.4 Vendor-Customer Relations

It would help end users if software has a "bill of materials" such that those using it could respond to a new threat in which some part of the software became a vector of attack. Users are sometimes prohibited by software licenses from publishing evaluations or comparisons with other tools. Georgetown University recently published a study of this issue [Klass16]. The study was sponsored by the Department of Homeland Security (DHS) Science & Technology Directorate (S&T), Cyber Security Division through the Security and Software Engineering Research Center (S^{2}ERC).

4.3.5 Standards

The development and adoption of standards and guidelines, as well as conformity assessment programs, are used across multiple industries to address quality. The US system of voluntary industry consensus standards allows for great flexibility to address needs. In some cases, the Government (federal, state or local) set regulatory standards and communities self-regulate. For example, the Institute of Electrical and Electronics Engineers (IEEE) released a Building Code for Medical Device Software Security in 2015 [Haigh15] and has launched an effort to develop a similar best practice document for energy and power distribution systems. Carl Landwehr proposed a "building code for building code" [Landwehr15]. Another example is NIST's *Framework for Improving Critical Infrastructure Cybersecurity* [Framework14].

4.3.6 Testing and Code Repositories

We explained the advantages of additional repositories of well-tested code in both Sections 2.1 and 2.4. In-depth testing of software is difficult and time-consuming, but necessary for key modules that are in common usage. Participants at the SPSQ workshop pointed out the value of both government and community-based efforts to test critical software. Code repositories promote code re-use and encourage organizations to test code by providing a location where the results can be published. Repositories could store some assurance level measure along with the code or even have built-in code measures tools. Repositories can also contain examples of projects with low bug densities, such as Tokeneer [Barnes06].

4.3.7 Threat Analysis

Threat analysis, sometimes called “threat modeling” or “risk analysis,” is a means of assessing risks or threats [Fundamental08]. Through threat analysis, software can be designed to avoid the introduction of some vulnerabilities and reduce the severity of others. For instance, one form of threat analysis is documenting attack surfaces to understand how adversaries might use interfaces to elevate privilege. Without performing threat analysis, preferably at both the architectural and design levels, software can contain vulnerabilities that might otherwise be avoided [Shostack14]. Architectural threat analysis can significantly increase the security robustness and resilience of the architecture of software and its high-level designs to dramatically reduce the number and severity of vulnerabilities [Diamant11].

4.4 Conclusions

The call for a dramatic reduction in software vulnerability is heard from multiple sources, including the 2016 Cybersecurity National Action Plan. This report has identified five approaches for achieving this goal. Each approach meets three criteria: 1) have a potential for dramatic improvement in software quality, 2) could make a difference in a three to seven-year time frame and 3) are technical activities. The identified approaches use multiple strategies:

- Stopping vulnerabilities before they occur, including improved methods for specifying and building software;
- Finding vulnerabilities, including better testing techniques and more efficient use of multiple testing methods and
- Reducing the impact of vulnerabilities by building architectures that are more resilient, so that vulnerabilities cannot be meaningfully exploited.

Formal Methods. Formal methods include multiple techniques based on mathematics and logic, ranging from parsing to type checking to correctness proofs to model-based development to correct-by-construction. While previously deemed too time-consuming, formal methods have become mainstream in many behind-the-scenes applications and show significant promise for both building better software and for supporting better testing.

System Level Security. System Level Security reduces the impact that vulnerabilities have. Operating system containers and microservices are already a significant part of the national information infrastructure. Given the clear manageability, cost and performance advantages of using them, it is reasonable to expect their use to continue to expand. Security-enhanced versions of these technologies, if adopted, can therefore have a widespread effect on the exploitation of software vulnerabilities throughout the National Information Infrastructure.

Additive Software Analysis. There are many types of software analysis – some are general and some target very specific vulnerabilities. The goal of additive software analysis is to be able to use multiple tools as part of an ecosystem. This will allow for increased growth and use of specialized software analysis tools and ability to gain a synergy between tools and techniques.

More Mature Domain-Specific Software Development Frameworks. The goal of this approach is to promote the use (and reuse) of well-tested, well-analyzed code, and thus to reduce the incidence of exploitable vulnerabilities.

Moving Target Defenses (MTD) and Automatic Software Diversity. This approach is a collection of techniques to vary the software's detailed structures and properties such that an attacker has much greater difficulty exploiting any vulnerability. The goal of automatic software diversity and MTD is to reduce an attacker's ability to exploit any vulnerabilities in the software, not to reduce the number of weaknesses in software.

A critical need for improving security is to have software with fewer and less exploitable vulnerabilities. The measures, techniques and approaches we have described will be able to do this. Higher quality software, however, does not get created in a vacuum. There must be a robust research infrastructure, education and training and customer demand. Higher quality software is a necessary step, but it is insufficient. A robust operation and maintenance agenda that spans a system's lifecycle is still needed.

4.5 Table of Acronyms

ACM	Association for Computing Machinery
AI	Artificial Intelligence
API	Application Program Interface
ASLR	Address Space Layout Randomization
AST	Abstract Syntax Tree
BLP	Bell-LaPadula
BSIMM	Building Security In Maturity Model
CAP	Cybersecurity Assurance Program
CEGAR	Counterexample-Guided Abstraction Refinement
CII	Core Infrastructure Initiative
CIL	Common Intermediate Language
CISQ	Consortium for IT Software Quality
CMU	Carnegie Mellon University
CPU	Central Processing Unit
CWE	Common Weakness Enumeration
DARPA	Defense Advanced Research Projects Agency
DHS	Department of Homeland Security
DoD	Department of Defense
DSL	Domain-Specific Language
ESAPI	Enterprise Security API
FAA	Federal Aviation Agency
FSSCC	Financial Services Sector Coordinating Council
GNU	Gnu’s Not Unix
GPU	Graphics Processor Unit
GUI	Graphical User Interface
ICT	Information Communications Technology
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
iFACTS	Interim Future Area Control Tools Support
I/O	Input/Output
IoC	Inversion of Control
IR	Intermediate Representation
ISO	International Organization for Standardization
ITS	Intrusion Tolerant Systems
KDM	Knowledge Discovery Metamodel
LoC	Lines of Code
ML	MetaLangauge
MTD	Moving Target Defense
NaN	Not a Number
NATS	National Air Traffic Service
NHTSA	National Highway Traffic Safety Administration
NICE	National Initiative for Cybersecurity Education

NICTA	National ICT Australia
NIST	National Institute of Standards and Technology
NITRD	Networking and Information Technology Research and Development
OBDD	Ordered Binary Decision Diagram
OS	Operating System
OSTP	Office of Science and Technology Policy
OWASP	Open Web Application Security Project
SAFES	Software Assurance Findings Expression Schema
SAL	Source code Annotation Language
SAT	Boolean Satisfiability
S^{2}ERC	Security and Software Engineering Research Center
SI	International System of Units
SMT	Satisfiability Modulo Theory
SPSQ	Software Productivity, Sustainability, and Quality
SSCA	Software and Supply Chain Assurance
S&T	Science and Technology
TCSEC	Trusted Computer Security Evaluation Criteria
TOIF	Tool Output Integration Framework

5 References

[Anderson72] James P. Anderson, “Computer Security Technology Planning Study,” Air Force ESD-TR-73-51, Vol. II, October 1972.

[Armstrong14] Robert C. Armstrong, Ratish J. Punnoose, Matthew H. Wong and Jackson R. Mayo, “Survey of Existing Tools for Formal Verification,” Sandia National Laboratories Report SAND2014-20533, December 2014. Available: http://prod.sandia.gov/techlib/access-control.cgi/2014/1420533.pdf Accessed 12 October 2016.

[Bakker14] Paul Bakker, “Providing assurance and trust in PolarSSL,” 8 May 2014, last modified 24 July 2015. Available: https://tls.mbed.org/tech-updates/blog/providing-assurance-and-trust-in-polarssl Accessed 21 June 2016.

[Barnes06] Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper and Bill Everett, “Engineering the Tokeneer Enclave Protection Software,” Proc. 1st IEEE International Symposium on Secure Software Engineering (ISSSE), March 2006. Available: http://www.adacore.com/uploads/technical-papers/issse2006tokeneer_altran.pdf Accessed 12 October 2016.

[Barnes13] John Barnes, “Safe and Secure Software: An Invitation to Ada 2012.” Available: http://www.adacore.com/knowledge/technical-papers/safe-and-secure-software-an-invitation-to-ada-2012 Accessed 13 October 2016.

[Barnett05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs and K. Rustan M. Leino, “Boogie: A Modular Reusable Verifier for Object-Oriented Programs,” in *Proc. 4th international conference on Formal Methods for Components and Objects (FMCO'05)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf and Willem-Paul de Roever, Eds. Springer, 2006, pp. 364-387, https://doi.org/10.1007/11804192_17.

[Barnum12] Sean Barnum, “Software Assurance Findings Expression Schema (SAFES) Overview,” January 2012. Available: https://www.mitre.org/publications/technical-papers/software-assurance-findings-expression-schema-safes-overview Accessed 8 September 2016.

[Barritt16] Keith Barritt, “3 Lessons: FDA/FTC Enforcement Against Mobile Medical Apps,” 14 January 2016. Available: http://www.meddeviceonline.com/doc/lessons-fda-ftc-enforcement-against-mobile-medical-apps-0001 Accessed 12 October 2016.

[Beck94] Kent Beck, “Simple Smalltalk testing: with Patterns,” *The Smalltalk Report*, 1994.

[Beizer90] Boris Beizer, *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold Co., New York, NY, ISBN: 0-442-20672-0.

[Bell76] D. E. Bell and L. J. La Padula, “Secure Computer System: Unified Exposition and Multics Interpretation,” Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, Technical Report No. ESD-TR-75-306, 1976.

[Biba77] K. J. Biba, “Integrity Considerations for Secure Computer systems,” Electronic Systems Division, AFSC, Hanscom AF Base, Bedford, MA, Technical Report ESD-TR-76-372, 1977. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a039324.pdf Accessed 12 October 2016.

[Bjørner16] Nikolaj Bjørner, “SMT Solvers: Foundations and Applications,” in *Dependable Software Systems Engineering*, Javier Esparza et. al., Eds. IOS Press, 2016, pp.24-32, https://doi.org/10.3233/978-1-61499-627-9-24.

[Black11a] Paul E. Black, Michael Kass, Michael Koo and Elizabeth Fong, “Source Code Security Analysis Tool Functional Specification Version 1.1,” NIST Special Publication (SP) 500-268 v1.1, February 2011, https://doi.org/10.6028/NIST.SP.500-268v1.1.

[Black11b] Paul E. Black, “Counting Bugs is Harder Than You Think,” in *Proc. 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011)*, Williamsburg, VA., 25-26 September 2011, pp. 1-9, https://doi.org/10.1109/SCAM.2011.24.

[Black16] Paul E. Black and Elizabeth Fong, “Report of the Workshop on Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV),” NIST Special Publication (SP) 500-320, October 2016, https://doi.org/10.6028/NIST.SP.500-320.

[Boebert85] W. E. Boebert and R. Y. Kain, “A Practical Alternative to Hierarchical Integrity Policies,” in *Proc. 8th National Computer Security Conference*, Gaithersburg, MD, 30 September-3 October 1985, pp.18-27. Available: http://csrc.nist.gov/publications/history/nissc/1985-8th-NCSC-proceedings.pdf Accessed 30 November 2016.

[Böhme08] Rainer Böhme and Felix C. Freiling, “On Metrics and Measurements,” in Irene Eusgeld, Felix Freiling and Ralf H. Reussner, Eds., in *Dependability Metrics*, *Lecture Notes in Computer Science*, Vol. 4909, Springer, 2008, pp. 7-13. https://doi.org/10.1007/978-3-540-68947-8_2.

[Bojanova16] Irena Bojanova, Paul E. Black, Yaacov Yesha and Yan Wu, “The Bugs Framework (BF): A Structured Approach to Express Bugs,” 2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2016), Vienna, Austria, 1-3 August 2016, https://doi.org/10.1109/QRS.2016.29.

[Boulanger15] Jean-Louis Boulanger, *CENELEC 50128 and IEC 62279 Standards*, John Wiley & Sons, 2015, https://doi.org/10.1002/9781119005056.

[Brooks95] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Addison-Wesley Professional, 1995, ISBN: 978-0201835953.

[Busoli07] Simone Busoli, “Inversion of Control and Dependency Injection with Castle Windsor Container - Part I,” 24 July 2007. Available: http://dotnetslackers.com/articles/designpatterns/InversionOfControlAndDependencyInjectionWithCastleWindsorContainerPart1.aspx Accessed 29 September 2016.

[Carter13] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman and Aaron Tomb, “SAW: The Software Analysis Workbench,” Proc. 2013 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '13), pp. 15-18, https://doi.org/10.1145/2527269.2527277.

[Chapman14] Roderick Chapman and Florian Schanda, “Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK,” in *Proc. Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014*. Gerwin Klein and Ruben Gamboa, Eds., *Lecture Notes in Computer Science*, Vol. 8558, Springer, 2014, pp. 17-26, https://doi.org/10.1007/978-3-319-08970-6_2.

[Chong05] Jennifer Chong, Partha Pal, Michael Atigetchi, Paul Rubel and Franklin Webber, “Survivability Architecture of a Mission Critical System: The DPASA Example,” in Proc. 21st Annual Computer Security Applications Conference (ACSAC 2005), December 2005, pp. 495-504, https://doi.org/10.1109/CSAC.2005.54.

[Claessen02] Koen Claessen and John Hughes, “Testing Monadic Programs with QuickCheck,” SIGPLAN Notices. Vol. 37, Issue 12, 2002, pp. 47–59, https://doi.org/10.1145/636517.636527.

[Clang] “clang: a C language family frontend for LLVM.” Available: http://clang.llvm.org/ Accessed 13 October 2016.

[CodeDx15] “Finding Software Vulnerabilities Before Hackers Do.” Available: https://codedx.com/wp-content/uploads/2015/10/AppSec101-FromCodeDx.pdf Accessed 8 September 2016.

[Corbato65] F. J. Corbató and V. A. Vyssotsky, “Introduction and Overview of the Multics System,” 1965 Fall Joint Computer Conference. Available: http://multicians.org/fjcc1.html Accessed 13 October 2016.

[Curtis16] Bill Curtis, private communication, 18 October 2016.

[DARPA16] DARPA, “Cyber Grand Challenge.” Available: https://www.cybergrandchallenge.com/ Accessed 21 October 2016.

[Diamant11] John Diamant, “Resilient Security Architecture: A Complementary Approach to Re ducing Vulnerabilities,” IEEE Security & Privacy, Vol. 9, Issue 4, July/August 2011, pp. 80-84, https://doi.org/10.1109/MSP.2011.88.

[Docker16] “Docker.” Available: https://www.docker.com/ Accessed 13 October 2016.

[Doyle16] Richard Doyle, “Formal Methods, including Model-Based Verification and Correct-By-Construction,” Dramatically Reducing Security Vulnerabilities sessions, Software and Supply Chain Assurance (SSCA) Working Group Summer 2016, McLean, Virginia, July 2016. Available: https://samate.nist.gov/docs/DRSV2016/SSCA_07_JPL_FormalMethods_Doyle.pdf Accessed 27 October 2016.

[duBousquet04] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat and J.-L. Lanet, “A case study in JML-based software validation,” in *Proc. 19th Int. IEEE Conf. on Automated Software Engineering (ASE'04)*, Linz, September 2004, pp. 294-297, https://doi.org/10.1109/ASE.2004.1342750.

[ECMA13] “ECMA-335 Common Language Infrastructure (CLI),” 6th ed., June 2012. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm Accessed 20 October 2016.

[FCRDSP16] *Federal Cybersecurity Research and Development Strategic Plan*, February 2016. Available: https://www.whitehouse.gov/sites/whitehouse.gov/files/documents/2016_Federal_Cybersecurity_Research_and_Development_Stratgeic_Plan.pdf Accessed 13 October 2016.

[Fisher16] Kathleen Fisher, John Launchbury and Raymond Richards, “The HACMS Program: Using Formal Methods to Eliminate Exploitable Bugs,” *Philosophical Transactions of the Royal Society A*, in submission as of October 2016. Presentation available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/fisher or slides available: https://www.usenix.org/sites/default/files/conference/protected-files/sec15_slides_fisher.pdf Accessed 25 October 2016.

[Flater15] David Flater, “Defensive code’s impact on software performance,” NIST Technical Note 1860, January 2015, https://doi.org/10.6028/NIST.TN.1860.

[Flater16] David Flater, Paul E. Black, Elizabeth Fong, Raghu Kacker, Vadim Okun, Stephen Wood and D. Richard Kuhn, “A Rational Foundation for Software Metrology,” NIST Internal Report (IR) 8101, January 2016. https://doi.org/10.6028/NIST.IR.8101.

[Fowler14] Martin Fowler, “Microservices: a definition of this new architectural term,” 25 March 2014. Available: http://martinfowler.com/articles/microservices.html Accessed 13 October 2016.

[FramaC] “What is Frama-C.” Available: http://frama-c.com/what_is.html Accessed 13 October 2016.

[Framework14] “Framework for Improving Critical Infrastructure Cybersecurity,” Version 1.0, NIST, 12 February 2014. Available: https://www.nist.gov/document-3766 Accessed 7 November 2016.

[Franz10] Michael Franz, “E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism,” Proc. New Security Paradigms Workshop (NSPW ’10), Concord, MA, 21–23 September 2010, pp. 7-16, https://doi.org/10.1145/1900546.1900550.

[FTC16] Federal Trade Commission, “Mobile Health App Developers: FTC Best Practices,” April 2016. Available: http://www.ftc.gov/tips-advice/business-center/guidance/mobile-health-app-developers-ftc-best-practices Accessed 13 October 2016.

[Fundamental08] *Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today*, Stacy Simpson, Ed., 8 October 2008. Available: http://www.safecode.org/publication/SAFECode_Dev_Practices1108.pdf Accessed 7 November 2016.

[Gabriel96] Richard P. Gabriel, *Patterns of Software: Tales from the Software Community*, Oxford Press, Oxford, 1996, ISBN: 0-19-5100269-X.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995, ISBN: 9788131700075.

[GCC16] "GCC, the GNU Compiler Collection." Available: https://gcc.gnu.org/ Accessed 13 October 2016.

[Goguen84] Joseph A. Goguen and Jose Meseguer, “Unwinding and Inference Control,” in *Proc. Symposium on Security and Privacy*, IEEE, 1984, pp. 75-86, ISBN: 0818605324.

[Grigg08] Ian Grigg, “The Market for Silver Bullets,” 2 March 2008. Available: http://iang.org/papers/market_for_silver_bullets.html Accessed 28 October 2016.

[Haigh15] Tom Haigh and Carl Landwehr, “Building Code for Medical Device Software Security,” IEEE Cyber Security, 2015. Available: https://www.computer.org/cms/CYBSI/docs/BCMDSS.pdf Accessed 27 October 2016.

[Heffley04] Jon Heffley and Pascal Meunier, “Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?” in *Proc. 37th Annual Hawaii International Conference on System Sciences (HICSS-04)*, 5-8 January 2004, Track 9, Volume 9, IEEE Computer Society, 2004, pp. 1-10, https://doi.org/10.1109/HICSS.2004.1265654.

[Hurd16] "GNU Hurd/hurd." Available: https://www.gnu.org/software/hurd/hurd.html Accessed 13 October 2016.

[ISO15939] "ISO/IEC 15939:2007 Systems and software engineering — Measurement process," 2007.

[ISO25010] "ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models," 2011.

[ISO25023] "ISO/IEC 25023:2016 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality," 2016.

[ISO25040] "ISO/IEC 25040:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Evaluation process," 2011.

[ISO26262-6] "ISO 26262-6:2011 Road Vehicles — Functional safety — Part 6: Product development at the software level," 2011.

[Iyer10] Vivek Iyer, Amit Kanitkar, Partha Dasgupta and Raghunathan Srinivasan, "Preventing Overflow Attacks by Memory Randomization," IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), November 2010. https://doi.org/10.1109/ISSRE.2010.22.

[Jézéquel97] Jean-Marc Jézéquel and Bertrand Meyer, "Design by Contract: The Lessons of Ariane," *IEEE Computer*, Vol. 30, Issue 1, January 1997, pp. 129-130, https://doi.org/10.1109/2.562936.

[Kastrinis13] George Kastrinis and Yannis Smaragdakis, "Hybrid Context-Sensitivity for Points-To Analysis," *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, 2013, pp. 423-434, https://doi.org/2499370.2462191.

[KDM15] Object Management Group, "Knowledge Discovery Metamodel (KDM)." Available: http://www.omg.org/technology/kdm Accessed 8 September 2016.

[Kiniry08] Joseph R. Kiniry and Daniel M. Zimmerman, "Secret Ninja Formal Methods," 15th International Symposium on Formal Methods (FM '08), Turku, Finland. May 2008. Available: http://verifiedgaming.org/publications_assets/KiniryZimmerman-FM08-SecretNinja.pdf Accessed 13 October 2016.

[Klass16] Gregory Klass and Eric Burger, "Vendor Truth Serum," Georgetown University. Available: https://s2erc.georgetown.edu/projects/vendortruthserum Accessed 19 September 2016.

[Klein14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski and Gernot Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Transactions on Computer Systems*, Vol. 32, Issue 1, February 2014, pp. 1-70, https://doi.org/10.1145/2560537.

[Knight12] John Knight, “Helix: Self-regenerative Architecture for the Incorruptible Enterprise,” Air Force Office of Scientific Research, AFRL-OSR-VA-TR-2012-1202, 13 November 2012. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a579086.pdf Accessed 20 October 2016.

[Kuhn10] Richard Kuhn, Raghu Kacker and Yu Lei, “Practical Combinatorial Testing”, NIST Special Publication (SP) 800-142, October 2010, https://doi.org/10.6028/NIST.SP.800-142.

[Kuhn13] D. Richard Kuhn, Itzel Dominguez Mendoza, Raghu N. Kacker and Yu Lei, “Combinatorial Coverage Measurement Concepts and Applications,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013)*, pp. 352-361, https://doi.org/10.1109/ICSTW.2013.77.

[Kuhn15] D. Richard Kuhn, Raghu N. Kacker and Yu Lei, “Measuring and specifying combinatorial coverage of test input configurations,” *Innovations in Systems and Software Engineering*, Vol. 12, Issue 4, December 2016, pp. 249-261, https://doi.org/10.1007/s11334-015-0266-2.

[Lampson04] Butler W. Lampson, “Software Components: Only the Giants Survive,” in *Computer Systems: Theory, Technology, and Application*, Karen I. B. Spaerck Jones and Andrew James Herbert, Eds., Springer, 2004, pp. 137-146, ISBN 978-0-387-20170-2. Available: http://research.microsoft.com/en-us/um/people/blampson/70-SoftwareComponents/70-SoftwareComponents.pdf Accessed 24 October 2016.

[Landwehr15] Carl Landwehr, “We Need a Building Code for Building Code,” *Communications of the ACM*, Vol. 58 No. 2, February 2015, pp. 24-26, https://doi.org/10.1145/2700341.

[Lemon13] Lemon, “Getting Started with LXC on an Ubuntu 13.04 VPS,” 6 August 2013. Available: https://www.digitalocean.com/community/tutorials/getting-started-with-lxc-on-an-ubuntu-13-04-vps Accessed 13 October 2016.

[Leroy06] Xavier Leroy, “Formal certification of a compiler back-end or: programming a compiler with a proof assistant,” Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’06), pp. 42-54, https://doi.org/10.1145/1111037.1111042.

[Lindholm15] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley, “The Java® Virtual Machine Specification,” Java SE 8 Edition, February 2015. Available: http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf Accessed 20 October 2016.

[LXC] “LXC," Ubuntu 16.04 Server Guide. Available: https://help.ubuntu.com/lts/serverguide/lxc.html Accessed 27 September 2016.

[Marien16] John R. Marien, Chair, and Robert A. Martin, Co-chair, “Suggested Language to Incorporate Software Assurance Department of Defense Contracts,” Department of Defense (DoD) Software Assurance (SwA) Community of Practice (CoP) Contract Language Working Group, February 2016. Available: http://www.acq.osd.mil/se/docs/2016-02-26-SwA-WorkingPapers.pdf Accessed 6 September 2016.

[McConnell04] Steve McConnell, *Code Complete*, 2nd Ed., Microsoft Press, Redmond, WA, 2004, ISBN: 0735619670.

[McIlroy68] M. D. McIlroy, “'Mass Produced' Software Components,” in *Software Engineering*, 1968 NATO Conference on Software Engineering, Garmisch, Germany, 7-11 October 1968, pp. 138-150. Available: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf Accessed 13 October 2016.

[Mell11] Peter Mell and Timothy Grance, “The NIST Definition of Cloud Computing,” NIST Special Publication (SP) 800-145, September 2011, https://doi.org/10.6028/NIST.SP.800-145.

[Mills87] Harlan D. Mills, Michael Dyer and Richard C. Linger, “Cleanroom Software Engineering,” *IEEE Software*, Vol. 4, Issue 5, 1987, pp. 19-25. Available: http://trace.tennessee.edu/utk_harlan/18/ Accessed 13 October 2016.

[Oberg99] James Oberg, “Why the Mars probe went off course,” *IEEE Spectrum*, Vol. 36, Issue 12, December 1999, pp. 34-39, https://doi.org/10.1109/6.809121.

[Okhravi13] H. Okhravi, M. A. Rabe, T. J. Mayberry, W. G. Leonard, T. R. Hobson, D. Bigelow and W. W. Streilein, “Survey of Cyber Moving Targets,” Massachusetts Institute of Technology Lincoln Laboratory. Technical Report 1166, 25 September 2013. Available: https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full_papers/2013_09_23_OkhraviH_TR_FP.pdf Accessed 13 October 2016.

[Okun04] Vadim Okun, Paul E. Black and Yaacov Yesha, “Comparison of Fault Classes in Specification-Based Testing,” *Information and Software Technology*, Elsevier, Vol. 46, Issue 8, June 2004, pp. 525-533, https://doi.org/10.1016/j.infsof.2003.10.003.

[Okun08] Vadim Okun, Romain Gaucher and Paul E. Black, Eds., “Static Analysis Tool Exposition (SATE) 2008,” NIST Special Publication (SP) 500-279, June 2009, https://doi.org/10.6028/NIST.SP.500-279.

[OMG16] Object Management Group, “Automated Source Code Maintainability Measure™ (ASCMM™) V1.0,” January 2016. Available: http://www.omg.org/spec/ASCMM/1.0 Accessed 12 October 2016.

[Ourghanlian14] Alain Ourghanlian, “Evaluation of static analysis tools used to assess software important to nuclear power plant safety,” *Nuclear Engineering and Technology: Special Issue on*

ISOFIC/ISSNP2014, Vol. 47, Issue 2, March 2015, pp. 212–218, https://doi.org/10.1016/j.net.2014.12.009.

[PaX01] "Design and Implementation of Address Space Layout randomization," https://pax.grsecurity.net/docs/aslr.txt, cited in "Address space layout randomization," Wikipedia. Available: https://en.wikipedia.org/wiki/Address_space_layout_randomization Accessed 15 September 2016.

[Perini16] Barti Perini, Stephen Shook and Girish Seshagiri, "Reducing Software Vulnerabilities – The Number One Goal for Every Software Development Organization, Team, and Individual," ISHIPI Technical Report, 22 July 2016.

[Pirsig74] Robert M. Pirsig, *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*, William Morrow & Company, New York 1974, ISBN: 9780688002305.

[Plan16] National Initiative for Cybersecurity Education (NICE), "Strategic Plan," April 2016. Available: http://csrc.nist.gov/nice/about/strategicplan.html Accessed 20 October 2016.

[Randimbivololona99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau and Dominique Schoen, "Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach," *FM '99 — Formal Methods: Proc. World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20-24, 1999, Volume II*, Jeannette M. Wing, Jim Woodcock and Jim Davies, Eds., *Lecture Notes in Computer Science*, Springer, Vol. 1709, 1999, pp. 1798-1815, https://doi.org/10.1007/3-540-48118-4.

[Rashid86] R. Rashid, "Threads of a New System," *Unix Review*, Vol. 4, No. 8, August 1986, pp. 37-49.

[Regehr15] John Regehr, "Comments on a Formal Verification of PolarSSL," 2015. Available: http://blog.regehr.org/archives/1261 Accessed 21 June 2016.

[Rice08] David Rice, *Geekonomics: The Real Cost of Insecure Software*, Addison-Wesley, 2008, ISBN: 978-0321477897.

[Rose16] "ROSE compiler infrastructure." Available: http://rosecompiler.org/ Accessed 8 September 2016.

[Rowe12] Jeff Rowe, Karl N. Levitt, Tufan Demir, Robert Erbacher, "Artificial Diversity as Maneuvers in a Control Theoretic Moving Target Defense," Proc. National Moving Target Research Symposium, Annapolis, MD, June 2012.

[Rushby05] John Rushby, "An Evidential Tool Bus," in *Proc. 7th international conference on Formal Methods and Software Engineering (ICFEM'05)*, Springer, 2005, p. 36, https://doi.org/10.1007/11576280_3.

[Saltzer75] Jerome H. Saltzer and Michael D. Shroeder, “The Protection of Information in Computer systems,” *Proc. IEEE* Vol. 63, Issue 9, September 1975, pp. 1278-1308, https://doi.org/10.1109/PROC.1975.9939.

[Shostack14] Adam Shostack, *Threat Modeling: Designing for Security*, Wiley and Sons, 2014, ISBN: 978-1-118-80999-0.

[SMTLIB15] “SMT-LIB: The Satisfiability Modulo Theories Library," 1 June 2015. Available: http://smtlib.cs.uiowa.edu Accessed 13 October 2016.

[Software16] “Software framework." Available: https://en.wikipedia.org/wiki/Software_framework Accessed 13 October 2016.

[Song08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam and Prateek Saxena, “BitBlaze: A New Approach to Computer Security via Binary Analysis,” *Proc. 4th International Conference on Information Systems Security (ICISS 2008), Hyderbad, India, 16-20 December 2008*, R. Sekar and Arun K. Pujari, Eds., *Lecture Notes in Computer Science*, Springer, Vol. 5352, 2008, pp. 1-25, ISBN 978-3-540-89862-7.

[Souyris09] Jean Souyris, Virginie Wiels, David Delmas and Hervé Delseny, “Formal Verification of Avionics Software Products,” *Proc. FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009*, Ana Cavalcanti and Dennis R. Dams, Eds., *Lecture Notes in Computer Science*, Springer, Vol. 5850, 2009, pp. 532–546, https://doi.org/10.1007/978-3-642-05089-3_34.

[Tschannen11] Julian Tschannen, Carlo A. Furia, Martin Nordio and Bertrand Meyer, “Verifying Eiffel Programs with Boogie,” *BOOGIE 2011: First International Workshop on Intermediate Verification Languages*, 2011. Available: https://arxiv.org/abs/1106.4700 Accessed 13 October 2016.

[TodoMVC16] “TodoMVC: Helping you select an MV* framework." Available: http://todomvc.com/ Accessed 13 October 2016.

[Tolerant07] “Tolerant Systems.” Available: http://www.tolerantsystems.org/ Accessed 13 October 2016.

[VCC13] VCC verifier. Available: https://vcc.codeplex.com/ Accessed 13 October 2016.

[Voas16a] Jeffrey Voas and Kim Schaffer, “Insights on Formal Methods in Cybersecurity," *IEEE Computer*, Vol. 49, Issue 5, May 2016, pp. 102–105, https://doi.org/10.1109/MC.2016.131.

[Voas16b] Jeffrey Voas and Kim Schaffer, “What Happened to Formal Methods for Security?”, *IEEE Computer*, Vol. 49, Issue 8, August 2016, pp. 70-79, https://doi.org/10.1109/MC.2016.228.

[Wayner15] Peter Wayner, “7 reasons why frameworks are the new programming languages," 30 March 2015. Available: http://www.infoworld.com/article/2902242/application-development/7-reasons-why-frameworks-are-the-new-programming-languages.html Accessed 13 October 2016.

[What] “What’s LXC?”, Available: https://linuxcontainers.org/lxc/introduction Accessed 13 October 2016.

[Whaley05] John Whaley, Dzintars Avots, Michael Carbin and Monica S. Lam, “Using Datalog with Binary Decision Diagrams for Program Analysis,” *3rd Asian Symposium on Programming Languages and Systems (ASPLAS)*, Tsukuba, Japan, 3-5 November 2005.

[Williams16] Chris Williams, “How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript,” 23 March 2016. Available: http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos Accessed 13 October 2016.

[Woodcock09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui and John Fitzgerald, “Formal Methods: Practice and Experience,” *ACM Computing Surveys*, Vol. 41, Issue 4, October 2009, pp. 1-36, https://doi.org/10.1145/1592434.1592436.

[Woodcock10] Jim Woodcock, Emine Gökçe Aydal and Rod Chapman, “The Tokeneer Experiments," in *Reflections on the Work of C.A.R. Hoare*, Cliff Jones, A. W. Roscoe and Kenneth R. Wood, Eds., July 2010, Chapter 17, pp. 405-430, https://doi.org/10.1007/978-1-84882-912-1_17.

[Woody14] Carol Woody, Robert Ellison and William Nichols, “Predicting Software Assurance Using Quality and Reliability Measures,” Technical Note CMU/SEI-2014-TN-026, December 2014. Available: http://resources.sei.cmu.edu/asset_files/technicalnote/2014_004_001_428597.pdf Accessed 13 October 2016.

[Woody16] Carol Woody, private communication, 17 October 2016.

[WSA04] “Web Services Architecture,” 11 February 2004. Available: https://www.w3.org/2002/ws/arch/ Accessed 27 October 2016.

[Wu11] Yan Wu, H Siy and Robin Gandhi, “Empirical results on the study of software vulnerabilities (NIER track),” in Proc. 33rd International Conference on Software Engineering (ICSE '11), Honolulu, Hawaii, 21-28 May 2011, pp. 964-967, https://doi.org/10.1145/1985793.1985960.

[Yang10] Jean Yang and Chris Hawblitzel, “Safe to the last instruction: automated verification of a type-safe operating system,” in *Proc. 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 99-110, https://doi.org/10.1145/1806596.1806610.

[Zhu97] Hong Zhu, Patrick A. V. Hall and John H. R. May, “Software unit test coverage and adequacy,” *ACM Computing Surveys*, Vol. 29, Issue 4, December 1997, pp. 366-427, https://doi.org/10.1145/267580.267590.