Draft 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

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

Draft 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

October 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
50 pages (October 2016)

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.

Public comment period: *October 4, 2016* through *October 18, 2016*

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: paul.black@nist.gov (SUBJECT="DRAFT NISTIR 8151 Comments")

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 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 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; security vulnerabilities; reduce software vulnerability.

Acknowledgements:

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

Effusive thanks 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 much of the material in Sect. 2.5 Moving Target Defense (MTD) and Artificial Diversity. Much of the wording is directly from a private communication from him.

Table of Contents

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 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 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) ….*

Since it is central to the purpose of this report, we define what we mean by "vulnerability." A vulnerability is a property of system security requirements, design, implementation or operation that could be accidentally triggered or intentionally exploited and result in a violation of desired system properties. A vulnerability is the result of one or more weaknesses in requirements, design, implementation or operation [Black11a]. This definition excludes

- operational problems, such as installing a program as world-readable or setting a trivial password for administrator access.
- insider malfeasance, such as exfiltration ala Snowden.
- functional bugs, such as the mixture of SI 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 by user names like "JoshuaCaleb." We exclude this vulnerability, because it is intentionally inserted. One assumes that a bad actor will fashion it to pass review/quality control processes.
- 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 the 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 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 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 meaningfully exploited.

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

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

1. Dramatic impact
2. 3 to 7-year timeframe
3. Technical activities

Dramatic. This means reducing exploitable vulnerabilities by two orders of magnitude. Estimates of software vulnerabilities are up to 25 errors per 1000 lines of code [McConnell04, page 521]. These approaches have been selected for the possibility of getting to 2.5 errors per 10 000 lines of code. The ability to measure whether an approach has a dramatic impact requires the ability to measure it. Measuring software quality is a difficult task. A parallel effort on improvements for measuring software vulnerabilities was pursued.

3 to 7-year timeframe. This timeframe was selected, because it is far enough out to make dramatic changes, based on existing techniques, but not having reached their full potential for impact. It is a timeframe that it is reasonable to speculate about. Beyond this timeframe, it is too difficult to predict what new technologies and techniques will be developed, potentially making their own set of dramatic changes on how IT is used. In the near future, the emphasis will be on implementing techniques that are already being deployed.

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 under Community Engagement. 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

This report excludes a discussion of vulnerabilities in firmware and 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, commercial and open source software. It covers software used for general use, mobile devices and embedded 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 METRICS

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 few vulnerabilities. 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.

1.3 METHODOLOGY

In order to produce the list of approaches, the Office of Science and Technology Policy asked NIST to lead a community-based effort. NIST consulted with multiple experts in the software assurance community including:

- Two Office of Science and Technology Policy (OSTP)-hosted inter-agency roundtables
- Half day session at the Software and Supply Chain Assurance Summer Forum

- Full day workshop on Software Measures and Metrics to Reduce Security Vulnerabilities
- Public comment 4-18 October 2016

1.4 REPORT ORGANIZATION

The report is organized into two major sections. The first enumerates technical approaches and the second addresses metrics.

Section 2 covers technical approaches in dealing with vulnerabilities in software. These include formal methods, such as rigorous static program analyses, model checkers and 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 artificial diversity. These include compile-time 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
- Rational for potential impact
- Further Reading, papers, other materials

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

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 3 to 7 year horizon. This list is not an exhaustive list, 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.

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. As the use of software exploded and programs grew so large that purely manual proofs were infeasible, formalized correctness arguments lost favor. In recent decades, developments such Moore’s law, multi-core processors and cloud computing make orders of magnitude more compute power readily available. Advances in algorithms for solving Boolean Satisfiability (SAT) problems, 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.

By the 1990s, formal methods had developed a bad 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. It is not that way anymore. 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. These are what Kiniry and Zimmerman call [Kiniry08] Secret Ninja Formal Methods: 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. Most proposed cryptographic protocols are now examined with model checkers for possible exploits. 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 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 possibly helpful formal method. Instead, we concentrate on a few that may contribute significantly in the medium term.

2.1.1 Rigorous 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 rigorous 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., source code, executables, requirements) may be statically analyzed. Source code analysis, however, is the most mature. Many tools have been developed to analyze software written in specific programming languages. One advantage of source code analysis is that the context of problems identified in source code can be communicated to software developers using a representation (the code itself) that is comprehensible to people. When other representations are analyzed, an additional step is required to render a problem into a form that people can first understand and then relate to a program under analysis.

According to Doyle’s assessment [Doyle16], rigorous static analysis is superior in terms of coverage, scalability and benefit for effort. A limitation is that it is difficult to specify some properties in available terms.

Formal methods have shown significant applicability in recent years. For example, the Tokeneer project shows [Barnes06, Woodcock10] that software can in some cases be developed with formal methods faster and cheaper and with fewer bugs than with traditional software development techniques. TrustInSoft used Frama-C to prove [Bakker14, Regehr15] the absence of a set of Common Weakness Enumeration (CWE) classes in PolarSSL, now known as mbed TLS. This approach is commonly used, and even mandated, in Europe for software in transportation and nuclear plant control.

These developments illustrate a few among 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 [Doyle16] is that model checkers can have excellent coverage and many properties can be represented. Since the effort required increases exponentially with problem size, there is always an effectual size limit, however. 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 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 there is no way 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.

For instance, the Open Web Application Security (OWASP) foundation coordinated a project to develop a shared application program interface (API), called Enterprise Security API (ESAPI). The ESAPI toolkit “encapsulate[s] the key security operations most applications need.”

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

2.1.4 Pragmas, Assertions, Pre- and Postconditions, Invariants, Properties, Contracts and Proof Carrying Code

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. Variations go by different terms, such as contracts, assertions, preconditions, postconditions and invariants. It cost programmers some thought to state exactly what is going on using a language similar to code expressions, but such statements help. These 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 track back from externally visible system failures. 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 precise 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. Analysis showed that a 16-bit integer could handle Ariane 4 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]

2.1.5 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 and others, such as design by refinement, aim to entirely avoid whole classes of vulnerabilities, since the developer rarely touches the code. Code synthesis like this is useful in fewer situations than other formal methods. Such models or specifications may also generate test suites or oracles. They may also be used to validate or monitor system operation.

According to Doyle's assessment [Doyle16], program synthesis has an "A+" in coverage, "B" in effort and properties, but "D" in scalability. When we can specify complete high-level models for entire systems, or even subsystems, we call them languages and cease to consider them unusual, but they represent a very substantial use of formal methods.

2.1.6 Maturity Level

Formal methods are today used (relatively invisibly) throughout the world. One of the most pervasive applications is the use of strong type checking 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 NICTA demonstrated [Klein14] the formal verification of the L4 microkernel comprising about 10 000 lines of C code.

2.1.7 Basis for Confidence

Assertions, and to a lesser extent, contracts, have been significantly adopted in high-quality software. Their gradual improvement to encompass more advanced condition and API checking is likely because they have already proven themselves in some developer communities. Many tools now perform static analysis. A natural progression is to promote more and more advanced forms of static analysis. Software proving based on techniques such as pre- and post-condition satisfaction and proof carrying code have seen initial adoption in critical software; they require more effort and cost, however, in some use cases they have been shown cost effective in the long run: fewer or no fixes to deployed systems.

2.1.8 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. As noted in [Woody14], unlike physical systems that wear out and eventually fail with greater frequently, software systems generate failures when they are incorrect and the flaws are triggered by environmental factors.

2.1.9 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 at http://prod.sandia.gov/techlib/access-control.cgi/2014/1420533.pdf

[Bjørner16] Nikolaj Bjørner, "SMT Solvers: Foundations and Applications", Dependable Software Systems Engineering, J. Esparza et. al. eds., pp 24-32, IOS Press, 2016. DOI: 10.3233/978-1-61499-627-9-24

[Boulanger12] "Industrial Use of Formal Methods: Formal Verification", Jean-Louis Boulanger (Ed), July 2012, Wiley-ISTE.

[Voas16a] Jeffrey Voas and Kim Schaffer, "Insights on Formal Methods in Cybersecurity", IEEE Computer 49(5):102 – 105, May 2016, DOI: 10.1109/MC.2016.131
A roundtable about formal methods with seven experts on formal methods.

[Voas16b] Jeffrey Voas and Kim Schaffer, (Insights, part 2), IEEE Computer August 2016

[Woodcock09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui and John Fitzgerald, "Formal Methods: Practice and Experience", ACM Computing Surveys, 41(4), October 2009, Article No. 19, DOI: 10.1145/1592434.1592436. Available at http://homepage.cs.uiowa.edu/~tinelli/classes/181/Fall14/Papers/Wood09.pdf

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, memory segments) to implement a computing utility that could protect information from unauthorized access by programs and users. 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. The noninterference model of [Goguen84] accounted for indirect information flows, also known as covert channels. Biba's integrity model expressed [Biba77] 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. The type enforcement model of [Boebert85] provided a table-based access control mechanism to allow data to be transformed only by pre-approved programs. These security policy 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 [Pal05]), 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 harms 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, 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, but a container can be launched in a fraction of a second on commodity hardware.

Container-based isolation can clearly reduce the impact of software vulnerabilities if the isolation is strong enough. Container configurations, however, are complex: they 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, e.g., 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 minimal resources that a program requires. Rather than trying to solve the problem in its full generality, one strategy is to develop analysis techniques/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. As explained in [Fowler14], microservices should implement individual business (or mission) capabilities, have independent refresh cycles, be relatively easy to replace and be programming-language agnostic. 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 kind of 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 pattern.

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.

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.

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, a growing number of microservice frameworks indicates that the technology is growing in its popularity and also that there is still room for enriching new microservices frameworks and for having the 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 Rational 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 wide-spread effect on the exploitation of software vulnerabilities.

2.2.6 Further Reading

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

[What] “What’s LXC?”, https://linuxcontainers.org/lxc/introduction/

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

2.3 Additive Software Analysis Techniques

Currently there are many different tools and techniques, both as open source and in commercial products, to analyze software, and they check for myriad problems. Many of them can by 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, like 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, say, 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 art 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 (hence, it must include standards), must provide building blocks to jumpstart new tool development (hence, it must include a framework) and must facilitate integration and interoperability among tools (hence, it must include 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 a standard, 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 kinds 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, invariants and so forth.
- 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 abstract syntax tree (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].

Compiler 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.

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, like C and C++, and higher-level object-oriented languages, like 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 like 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 is to integrate a checker as a plugin into an existing compiler toolchain. Modern compiler frameworks like 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. 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 Section 2.4.

2.3.3 Combining 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 class (INJ) bugs [Bojanova16]. Adding a tool to check for deadlocks could give the programmer more information.

The second case is confirmatory. The programmer may have two different heuristics to find faulty operation (FOP) 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.

The third case of additive software analysis is synergy. A research group with expertise in formal reasoning about memory use and data structures can 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 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, though languages like Boogie have been successfully used as the engine by several research groups for building checkers for diverse languages like C [VCC13], 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). Efforts to consolidate the output of tools, such as Tool Output Integration Framework (TOIF), Software Assurance Findings Expression Schema (SAFES) [Barnum12] and Code Dx [CodeDx15], already implicitly indicate classes of kinds of useful knowledge about software.

2.3.5 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.

2.3.6 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 that 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, 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 [Kastrinis14].

2.3.7 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 Int'l Conf. on Software Quality, Reliability, and Security (QRS 2016), Vienna, Austria, August 1-3, 2016. Available at https://samate.nist.gov/BF, accessed 12 September 2016.

[Kastrinis14] G. Kastrinis and Y. Smaragdakis, “Hybrid Context-Sensitivity for Points-To Analysis,” *Proc. Conference on Programming Language Design and Implementation (PLDI),* 2014

[Rushby05] John Rushby, “An Evidential Tool Bus,” *Proc. International Conference on Formal Engineering Methods,* 2005.

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, for example: standalone programs, services, micro-services, modules, plugins, libraries of functions, frameworks, 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, importantly, also defines a software architecture (including default behavior and flow of control) for programs built using it. A domain-specific framework furthermore includes domain knowledge, e.g., GUI building, parsing, Web applications, multimedia, scheduling. 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, 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 is difficult to achieve. Specifically, in order 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. 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.

Learning new Frameworks. Brooks said [Brooks95] that software embodies both "essential" and "accidental" information. 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 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, and 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 left-pad incident of 2016 illustrates the danger. The heavily-used Node Package Manager maintains numerous packages that 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, like 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 [Lampson04], each component may have a distinct “world view” and the composition of n components can result in n^2 interactions.

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 scale, 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 Compositional Testing

Advanced testing approaches hold 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. Recent advances in the measurement of traditional test suite coverage provide an opportunity to compare frameworks. Combinatorial testing [Kuhn10] has been used to improve on black box fuzz testing as well as to test alternate software configurations. 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 supporting techniques listed in this section 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 Rational 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", https://en.wikipedia.org/wiki/Software_framework

[TodoMVC16] "TodoMVC: Helping you select an MV* framework", http://todomvc.com/

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

2.5 Moving Target Defenses (MTD) and Artificial Diversity

This approach is a collection of techniques to vary software's detailed structures and properties such that an attacker has much greater difficulty exploiting any vulnerability. To illustrate, consider one early, widely-used technique in this family: Address Space Layout Randomization (ASLR), invented in 2001 by the PaX Team [PaX01]. When a program requests a buffer, the easiest thing is to return the next available chunk of memory. This puts buffers in the same relative location. Knowing this, an attacker can exploit a buffer overflow weakness (BOF) [Bojanova16] in one buffer to, say, read the password that is in another buffer that is always 384 bytes beyond it. ASLR puts buffers in different (unpredictable) relative locations, so that the above exploit is much harder.

The goal of artificial 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.

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 compute 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 resource invested, the higher the amount of assurance. The difference is that static analysis provides assurance that the software does not contain vulnerabilities of specific types, while MTD provides assurance that weaknesses of any type are expensive to exploit.

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, ASLR, 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 compute 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.

2.5.2 System or Network Techniques

Some techniques at the system or network level are network address space randomization and protocol diversity. 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 address or a shared secret key, so an application can authenticate and get current information.

2.5.3 Operating System 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 C of memory management APIs, a set B of process services, a set D of networking functions and a set A of 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 in order to succeed.

2.5.4 Maturity Level

Some moving target defenses are the default in many operating systems and compilers today. There is intense research and entire conferences to understand limitations, costs and benefits of current techniques and develop 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, September 2013. Available at https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full_papers/2013_09_23_OkhraviH_TR_FP.pdf Accessed 15 September 2015.

3 Measures and Metrics

This section deals with metrics, measures, assessments, appraisals, judgements, evaluations, etc. in the broadest sense. Hence, code reviews and software testing have a place in this section. We have three areas of concern. First, encouraging the use of metrics. All the extraordinary metrics in the world do not help if nobody uses them. Also, nobody *can* act on metrics if the metrics are not produced and available. The Federal Government might motivate and encourage the use of software product metrics. Vehicles include procurement, contracting, liability, insurance and also standards as explained in Sect. 4.3. Software can also benefit from the programs and criteria of third-party, non-governmental organizations. Some possibilities are Underwriter's Laboratory 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 metrics, which is the second area.

The second area, process metrics includes hours of effort, number of changes with no acceptance test defects or 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 doesn't allow for training, the number of vulnerabilities is likely to be much higher. Other examples are software measures that indicate how much a new process step helps compared to the former practice or metrics 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 metrics that are most applicable to their circumstance. We do not discuss process metrics further.

The final area of concern is metrics 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 or 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 and metrics to significantly reduce vulnerabilities. The web site is https://samate.nist.gov/SwMM-RSV2016.html. They called for short position statements, 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-XXXX. 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 metrics and measures. A metric is a simple, basic assessment or count with a clear value. A measure, on the other hand, is derived from other metrics and measures. Measures are often surrogates for properties that we would like to be able to determine. For instance, number of buffer overflow weaknesses is a metric with a reasonably clear definition. In contrast, code security is a measure that is only loosely related to the number of buffer overflows. The absence of flaws does not indicate the presence of excellence.

3.1 A Taxonomy of Software Metrics

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

Software metrics may be divided into two broad categories as to whether they are low-level or high-level. Low-level metrics are generally widely applicable. High-level metrics, in contrast, deal with the relation between the program, as an object, and the developer or user, as a sentient subject. It is in this interaction between object and subject that quality arises, as Pirsig said. [Pirsig74] Analogously to low- and high-level metrics, 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's a possibility of a buffer overflow (BOF) [Bojanova16] given particular inputs. There is no need to refer to a specification, requirement or security policy to determine whether a buffer overflow is possible.

On the other hand, high-level vulnerabilities cannot be discerned solely by reference to the code. A human reviewer or a static analyzer must refer to 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 attempt 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 metrics is most apparent in testing. Test metrics conceptually have two parts: test generation or selection and test result evaluation. Test metrics generally answer the question, how much of the program (internal) or the input space (external) has been exercised? Test case generation is necessarily static, while evaluation is usually Θdynamic, that is, based on the result of executions. In many test metrics, the two parts are tied to each other. They 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.

The third dimension is the point of view, either external or internal. External metrics are typically behavioral conformance to specifications, requirements or constraints. They are often referred to as “black box” or behavioral. These metrics are particularly useful for acceptance testing and estimating user or mission satisfaction. It matters little how well the program functions or is structured internally if it does not fulfil its purpose. In contrast, internal or structural metrics primarily deal with, or are informed by, the code’s architecture, implementation and fine-grained operation. Metrics in this class are related to qualities such as maintainability, portability, elegance and potential. For instance, external 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 internal and external metrics. External metrics, such as boundary value analysis [Beizer90] and combinatorial testing [Kuhn10], consider the behavioral or specification in computing how much has been tested or what has not been tested. On the other hand, internal metrics include counts of the number of blocks, mutation adequacy [Okun04], and path coverage metrics [Zhu97]. The two approaches are complementary. External testing can find missing features. Internal 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 metrics conceptually divides them into three types. The first is presence (or absence) of particular weaknesses such as buffer overflow (BOF) or injection (INJ) [Bojanova16]. Note that the absence of flaws does not indicate, say, resilient architecture. The second type is quality metrics that directly measure that code, or parts of it, is excellent. However, we only have proxies for “quality,” like maintainability, portability or the presence of assertions. The third type is conformance to specification or correctness. This third type of metric 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 metric or measure that guarantees excellent code.

3.2 Software Assurance: The Object of Software Metrics

Software assurance, that is, our assurance that software will behave as it should, comes from three broad sources. The first is the development process. If software is developed by a team with clear requirements, are well trained and who have demonstrated an ability to build good software with low vulnerability rates, then we have confidence or assurance that software that they produce is likely to be 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, give it few system privileges, then have other programs monitor the execution. Then if any vulnerabilities are triggered, the damage to the system is controlled.

With research we may be able to give detail to the mathematical formula that expresses our assurance: $A = f(p, s, e)$ 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.

3.3 Software Metrology

To have a coherent, broadly useful system of metrics, one must have a solid theoretical foundation. That is, a philosophy of software measurement. 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 metrics have well known theoretical limitations, too. 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 metrics 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 satisfactory 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 metric might apply only to program that do not have perverse structures. A metric may still be useful even if it doesn't apply to programs consisting solely of millions of conditional go-to statements with seemingly arbitrary computations interspersed. Nobody (should) write programs like that. Finally, society may decide that for certain applications, we will only build measurable software. Architects are not allowed to design building with arbitrary structures. They must run analyses showing that the design withstands expected loads and forces. Instead of writing some software and trying to show that it works, the expectation might change to only writing 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 sue 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 metrics. For example, Function Points 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 you want and the features you have. 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. [ASCMM16] It is not terribly 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. On the other hand, software metrology has a distinct gap. We want to determine measure high-level properties such as quality, maintainability and security, but we do not have precise definitions of those, and therefore cannot measure those directly. We can, however, measure many properties which 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 Metrics

As much as good process is essential to the production of code with few vulnerabilities, the ultimate 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, which includes testing and checking, must be include in *all* phases of software development. Except for ambitious approaches like Clean Room, 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 metrics may be the wrong emphasis to reduce software vulnerabilities. Such metrics may fade in emphasis as other software metrics have, for example cohesion and McCabe Cyclomatic Complexity. Perhaps the best approach is a “Clean Room” approach, in which metrics inform a decision to accept or reject and do not purport to establish an absolute certification of freedom from errors.

3.4.1 Existing Metrics

There are hundreds of proposed software metrics and 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 only have moderate correlation with 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 languages, which allow a programmer to express functionality more succinctly, will 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, say not releasing a resource after use, is that one problem or several? Rather than being the exception, these kinds of complication are the rule in software [Okun08].

For any realistic program, it is infeasible to try every single possible input. Instead, one must choose a metric that spans the entire space. Some of these metrics are combinatorial input metrics [Kuhn10], mutation adequacy [Okun04], path coverage metrics [Zhu97] and boundary value analysis [Beizer90].

There are far too many proposed measures to evaluate or even list here. We can state that, as alluded to above, metrics and 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 workshop presentations, 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, like Less Hatton's SaferC. 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.

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

3.4.3 Metrics and Measures of Binaries and Executables

Some workshop participants were of the opinion that there is a significant need for metrics and 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 subtle 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.

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," January 2016. Available at http://www.meddeviceonline.com/doc/lessons-fda-ftc-enforcement-against-mobile-medical-apps-0001

[FTC16] "Mobile Health App Developers: FTC Best Practices," April 2016. Available at http://www.ftc.gov/tips-advice/business-center/guidance/mobile-health-app-developers-ftc-best-practices

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

4 Summary and Community Engagement

In response to the February 2016 Federal Cybersecurity Research and Development Strategic Plan, NIST was asked 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 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, improved scanning during development and use 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. The 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. Many organizations use bug bounty programs to incentive the research community to find and notify organizations about bugs.

4.1.2 Research Infrastructure

There is a need for repositories of data related to secure software. Several very successful repositories exist, 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? 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. This is the primary mechanism how new approaches are transitioned 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, 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.

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 it has been received. Improved metrics that are customer-focused are needed as are other policy and economic approaches. Policy and economic approaches are outside the scope of this report, but 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 Government 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 focus on low bidder 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 felt that companies developing software should be contractually liable for vulnerabilities discovered after delivery. Many participants 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^2ERC).

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 or state/local) set regulatory standards and communities often self-regulate.

4.3.6 Code Repositories

We explained the need for additional repositories of well-tested code in both Sections 2.1 and 2.4. Code repositories promote code re-use and encourage organizations to test code by providing a location where the results can be published. Repositories can also contain examples of low bug densities projects such as Tokeneer. [Barnes06]

4.4 Conclusion

The call for a dramatic reduction in software vulnerability is heard from multiple sources, including the 2015 Cybersecurity 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 timeframe and 3) are technical activities. The identified approaches use multiple strategies:

- Stopping vulnerabilities before they occur generally including improved methods for specifying and building software.
- Finding vulnerabilities including better testing techniques and more efficient use of multiple testing methods.
- Reducing the impact of vulnerabilities by building architectures that are more resilient, so that vulnerabilities can’t 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 wide-spread 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 Artificial 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 artificial diversity and moving target defense (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, though, does not get created in a vacuum. There must be a robust research infrastructure, education and training, and customer pull. 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.

5 References

[Anderson72] James P. Anderson, “Computer Security Technology Planning Study,” Air force ESD-TR-73-51, Vol II, Oct. 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. http://prod.sandia.gov/techlib/access-control.cgi/2014/1420533.pdf

[ASCMM16] “Automated Source Code Maintainability Measure™ (ASCMM™) V1.0”, http://www.omg.org/spec/ASCMM/1.0, January 2016

[Bakker14] Paul Bakker, “Providing assurance and trust in PolarSSL”, 2014 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 at http://www.adacore.com/uploads/technical-papers/issse2006tokeneer_altran.pdf

[Barnett05] M. Barnett, B.E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, “Boogie: A Modular Reusable Verifier for Object-Oriented Programs,” *Proc. International Symposium on Formal Methods for Components and Objects (FMCO)*, 2005

[Barnum12] Sean Barnum, “Software Assurance Findings Expression Schema (SAFES) Overview,” January 2012. Available at 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,” January 2016. Available at http://www.meddeviceonline.com/doc/lessons-fda-ftc-enforcement-against-mobile-medical-apps-0001

[Beck94] K. 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. Lapadula, “Secure Computer System: Unified Exposition and Multics Interpretation,” Technical Report No. ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976.

[Biba77] K.J. Biba, "Integrity Considerations for Secure Computer systems," USAF Electronic Systems Division, Bedford, MA, ESD-TR-76-372, 1977.

[Bjørner16] Nikolaj Bjørner, "SMT Solvers: Foundations and Applications", in *Dependable Software Systems Engineering*, J. Esparza et. al. eds., pp 24-32, IOS Press, 2016. DOI: 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, DOI: 10.6028/NIST.SP.500-268v1.1

[Black11b] Paul E. Black, "Counting Bugs is Harder Than You Think," 11th IEEE Int'l Working Conference on Source Code Analysis and Manipulation (SCAM 2011), September 2011, Williamsburg, VA.

[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-xxx, October 2016.

[Boebert85] W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," Proc. 8th National Computer Security Conference, Gaithersburg, MD, p.18, 1985.

[Bojanova16] Irena Bojanova, Paul E. Black, Yaacov Yesha and Yan Wu, "The Bugs Framework (BF): A Structured Approach to Express Bugs," 2016 IEEE Int'l Conf. on Software Quality, Reliability, and Security (QRS 2016), Vienna, Austria, August 1-3, 2016. Available at https://samate.nist.gov/BF, accessed 12 September 2016.

[Boulanger12] Jean-Louis Boulanger (Ed.) Industrial Use of Formal Methods: Formal Verification, July 2012, Wiley-ISTE.

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

[Brooks95] F. Brooks, The Mythical Man-Month, Anniversary edition with 4 new chapters, Addison-Wesley, 1995.

[Clang] "Clang: A C language family frontend for LLVM," http://clang.llvm.org/

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

[Corbato65] F.J. Corbato and V.A. Vyssotsky, "Introduction and Overview of the Multics System," 1965 Fall Joint Computer Conference. Available at http://multicians.org/fjcc1.html

[Docker16] "Docker," https://www.docker.com/

[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 at https://samate.nist.gov/docs/DRSV2016/SSCA_07_JPL_FormalMethods_Doyle.pdf

[FCRDSP16] Federal Cybersecurity Research and Development Strategic Plan. Available at http://www.whitehouse.gov/sites/whitehouse.gov/files/documents/2016_Federal_Cybersecurity_Research_and_Development_Strategic_Plan.pdf

[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. Available at https://doi.org/10.6028/NIST.IR.8101

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

[FramaC] "What is Frama-C", http://frama-c.com/what_is.html

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

[Gabriel96] Richard P. Gabriel, "Patterns of Software: Tales from the Software Community," Oxford Press, 1996.

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

[GCC16] The GNU Compiler Collection, available at https://gcc.gnu.org/

[Goguen84] J.A. Goguen and J. Meseguer, "Unwinding and Inference Control," Proc. 1984 IEEE Symposium on Security and Privacy, 1984.

[Hurd16] "Hurd", Web Services Architecture Workshop Group, https://www.gnu.org/software/hurd/hurd.html, 2016.

[Jézéquel97] Jean-Marc Jézéquel and Bertrand Meyer, "Design by Contract: the Lesson of Ariane,", IEEE Computer, 30(1):129-130, January 1997. DOI: 10.1109/2.562936

[Kastrinis14] G. Kastrinis and Y. Smaragdakis, "Hybrid Context-Sensitivity for Points-To Analysis," *Proc. Conference on Programming Language Design and Implementation (PLDI),* 2014.

[KDM15] "Knowledge Discovery Metamodel (KDM)," available at http://www.omg.org/technology/kdm/, accessed 8 September 2016.

[Kiniry08] Joseph Kiniry and Daniel Zimmerman, "Secret Ninja Formal Methods," 15th Int'l Symp. on Formal Methods (FM'08). Turku, Finland. May, 2008.

[Klass16] Gregory Klass and Eric Burger, "Vendor Truth Serum", Georgetown University, 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," NICTA and UNSW, Sydney, Australia, ACM Transactions on Computer Systems, Vol. 32, No. 1, Article 2, Publication date: February 2014.

[Kuhn10] Richard Kuhn, Raghu Kacker and Y. Lei, "Practical Combinatorial Testing", NIST Special Publication 800-142, Oct. 2010. Available at http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

[Lampson04] B.W. Lampson, "Software components: Only the Giants survive," Computer Systems: Theory, Technology, and Application, Springer, 2004.

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

[LXC] "LXC", Ubuntu 16.04 Server Guide, 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 at 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, 2004. ISBN: 0735619670

[Mcilroy68] D. McIlroy, "Mass Produced Software Components," 1968 NATO Conference on Software Engineering.

[Oberg99] James Oberg, “Why the Mars probe went off course,” IEEE Spectrum, 36(12):34-39, December 1999. DOI: 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, September 2013, Available https://www.ll.mit.edu/mission/cybersec/publications/publications:files/full_papers/2013_09_23_OkhraviH_TR_FP.pdf Accessed 15 September 2015.

[Okun04] Vadim Okun, Paul E. Black and Yaacov Yesha, “Comparison of Fault Classes in Specification-Based Testing,” Information and Software Technology, Elsevier, 46(8):525-533, June 2004.

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

[Pal05] Partha Pal, Michael Atigetchi, Jennifer Chong, Franklin Webber and Paul Rubel, “Survivability Architecture of a Mission Critical System: The DPASA Example”, 21st Annual Computer Security Applications Conference (ACSAC 2005), pages 495-504, ISSN: 1063-9527, ISBN: 0-7695-2461-3, DOI: 10.1109/CSAC.2005.54, December 2005.

[PaX01] “Design and Implementation of Address Space Layout randomization,” https://pax.grsecurity.net/docs/aslr.txt, cited in “Address space layout randomization,” Wikipedia. Available at https://en.wikipedia.org/wiki/Address_space_layout_randomization Access 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, 1974.

[Rashid86] R. Rashid, “Threads of a New System,” Unix Review, Vol 4, No. 8, August 1986.

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

[Rose16] “ROSE compiler infrastructure,” http://rosecompiler.org/, accessed 8 September 2016.

[Rushby05] John Rushby, “An Evidential Tool Bus,” *Proc. International Conference on Formal Engineering Methods*, 2005.

[Saltzer75] J. Slatzer and M. Shroeder, “The Protection of Information in Computer systems,” Proc. IEEE vol. 63, issue 9, p. 1278-1308, September 1975.

[SMTLIB15] “SMT-LIB: The Satisfiability Modulo Theories Library”, http://smtlib.cs.uiowa.edu/, June 2015.

[Software16] “Software framework”, https://en.wikipedia.org/wiki/Software_framework

[Tschannen11] J. Tschannen, C. A. Furia, M. Nordio and B. Meyer, “Verifying Eiffel Programs with Boogie,” *Boogie: First International Workshop on Intermediate Verification Languages*, 2011.

[TodoMVC16] “TodoMVC: Helping you select an MV* framework”, http://todomvc.com/

[Tolerant07] “Tolerant Systems,” http://www.tolerantsystems.org/

[VCC13] VCC verifier, available at https://vcc.codeplex.com/

[Voas16a] Jeffrey Voas and Kim Schaffer, “Insights on Formal Methods in Cybersecurity”, IEEE Computer 49(5):102–105, May 2016, DOI: 10.1109/MC.2016.131

[Voas16b] Jeffrey Voas and Kim Schaffer, (Insights, part 2), IEEE Computer, August 2016.

[Wayner15] Peter Wayner, “7 reasons why frameworks are the new programming languages”, http://www.infoworld.com/article/2902242/application-development/7-reasons-why-frameworks-are-the-new-programming-languages.html, March 2015.

[What] “What’s LXC?”, https://linuxcontainers.org/lxc/introduction/

[Whaley05] J. Whaley, D. Avots, M. Carbin and M. S. Lam, “Using Datalog with Binary Decision Diagrams for Program Analysis,” *Proc. Programming Languages and Systems (ASPLAS)*, 2005.

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

[Woodcock09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui and John Fitzgerald, “Formal Methods: Practice and Experience,” ACM Computing Surveys, 41(4), October 2009, Article No. 19, DOI: 10.1145/1592434.1592436. Available at http://homepage.cs.uiowa.edu/~tinelli/classes/181/Fall14/Papers/Wood09.pdf

[Woodcock10] Jim Woodcock, Emine Gökçe Aydal and Rod Chapman, “The Tokeneer Experiments”, in Reflections on the Work of C.A.R. Hoare, History of Computing, Chapter 17, pp 405-430, July 2010, DOI: 10.1007/978-1-84882-912-1_17.

[Woody14] Carol Woody, R. Ellison and W. Nichols, "Predicting Software Assurance Using Quality and Reliability Measures," CMU/SEI-2014-TN-026, Dec. 2014.

[WSA04] "Web Services Architecture", https://www.w3.org/2002/ws/arch/

[Wu11] Yan Wu, H. Siy and Robin A. Gandhi, "New Ideas and Emerging Results Track: Empirical Results on the Study of Software Vulnerabilities," New Ideas and Emerging Results Track at the 33rd International Conference on Software Engineering (ICSE 2011), Honolulu, Hawaii, May 21-28, 2011.

[Yang10] J. Yang and C. 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.

[Zhu97] Hong Zhu, Patrick A. V. Hall and John H. R. May, "Software Unit Test Coverage and Adequacy," ACM Computing Surveys (CSUR), 29(4): 366-427, December 1997, DOI: 10.1145/267580.267590