The National Need for Software Understanding

The Present Crisis, Technical Capability Gaps, and Path Forward

Douglas Ghormley
Tod Amon
Christopher Harrison
Tim Loffredo

Sandia National Laboratories

Revision 1.0
March 25, 2025

Web: https://suns.sandia.gov/
Email: suns@sandia.gov

Executive Summary

Our nation is presently facing extensive, unmeasurable risk to national security and critical infrastructure (NS&CI) missions through widespread dependence on largely inscrutable third-party and legacy software. In recent decades, software has been integrated into every facet of government and society, including NS&CI missions. Historical choices on regarding software have resulted in economic prosperity and opportunity, but also a tremendous gap between our total dependence on software in NS&CI missions and our extremely limited capability to understand and validate it.

Despite rigorous testing, software typically contains unexpected behavior that can imperil the missions that rely upon it. Focused efforts to improve development practices are laudable but insufficient to address the vast volume of third-party and legacy software already in use, let alone the accelerating pace of new and updated software requiring mission analysis. Ideally, to ensure mission success, mission owners would routinely analyze all mission-relevant software to seek technical evidence about its potential behavior before putting it into operation. Unfortunately, the technical tools and capabilities to do so do not exist. The broad set of mission questions to address, the vast array of software upon which we depend, and the speed at which answers are needed, together pose a software understanding challenge that exceeds current technical capabilities.

Every U.S. government (USG) agency faces this same challenge, yet there is no coordinated activity to create the necessary capabilities. The technical analysis capabilities needed by different agencies and missions share extensive commonality yet are typically developed in isolation, squandering the potential for a larger return on investment for the broader government.

The net result is that our nation faces potential catastrophe as our vital missions and most sensitive systems rely on software components that we cannot adequately characterize, leading to extensive, unknown, and unbounded mission risk. Our confidence today in NS&CI mission software is largely based on unsubstantiated assumption, not on reliable, technical evidence.

In March 2023, five government representatives co-convened the “Software Understanding for National Security (SUNS) 2023 Workshop”, which brought together mission stakeholders and technical subject matter experts (SMEs) from 18 government groups and Federally Funded Research and Development Centers (FFRDCs) to discuss these issues. The workshop considered 1) the importance of the problem, 2) the feasibility of the technical capabilities required to meet NS&CI mission needs, and 3) the challenges and possible paths forward for producing them.

Significantly, a vast majority of the SMEs participating expressed opinions consistent with the following key results from the workshop:

1. **A 10-100x or more improvement in software understanding capabilities is possible, but many non-technical issues are currently inhibiting progress**.
2. **A unified national effort is needed to revolutionize our software understanding capabilities to meet current and future NS&CI mission needs.**

In January 2025, four U.S. government agencies coauthored a report on “Closing the Software Understanding Gap,” calling for decisive and coordinated action across the U.S. Government regarding these issues.

This report explains the problem of software understanding for national security and critical infrastructure missions, discusses the shortcomings of traditional investment approaches, documents the findings of the SUNS 2023 Workshop, and concludes with recommendations. For a synopsis of the 2023 Workshop, FAQs, and updates see https://suns.sandia.gov/.

TABLE OF CONTENTS

EXECUTIVE SUMMARY 1

LIST OF FIGURES 4

LIST OF TABLES 4

ACRONYMS AND DEFINITIONS 4

DOCUMENT REVISIONS 5

1 INTRODUCTION 6
- 1.1 Acknowledgments 8

2 THE PROBLEM: UNBOUNDED MISSION RISK FROM INSCRUTABLE SOFTWARE 9
- 2.1 Unexpected Behavior in Software 11
- 2.2 Lack of Adequate Solutions 13
- 2.3 An Accelerating Problem 14
- 2.4 Conclusions Regarding the Software Understanding Problem 16

3 THE SOFTWARE UNDERSTANDING GAP 17
- 3.1 Software Understanding Defined 18
- 3.2 The Ideal State of Rigorous, Reliable, Rapid, and Repeatable Software Understanding 18
- 3.3 The Current State of Software Understanding Alternatives 24
- 3.4 Conclusions Regarding the Software Understanding Gap 31

4 THE HISTORY AND CHALLENGE OF UNDERSTANDING SOFTWARE 32
- 4.1 Historical Choices that Drive the Software Understanding Gap 33
- 4.2 Software Characteristics that Drive the Software Understanding Gap 39
- 4.3 Implications and Observations from Software History and Characteristics 45
- 4.4 Conclusions and the Software Understanding Gap 49

5 SOFTWARE UNDERSTANDING CASE STUDIES 50
- 5.1 Volkswagen Emissions Scandal 50
- 5.2 SolarWinds 52
- 5.3 Log4j 55
- 5.4 Node-ipc 57
- 5.5 Conclusion 58

6 OPTIONS FOR ADDRESSING THE SOFTWARE UNDERSTANDING GAP 59
- 6.1 Accept the Software Understanding Gap as Inevitable 59
- 6.2 Redouble Efforts in Software Understanding Alternatives 60
- 6.3 Pursue Automated, Rigorous Technical Solutions to Software Understanding 62
- 6.4 A National Crossroads 63

7 THE SUNS 2023 WORKSHOP AND RESULTS 64
- 7.1 Editorial Note 65
- 7.2 Key Questions 65
- 7.3 Participants 65
- 7.4 Workshop Structure 67
- 7.5 Key Outcomes 67
- 7.6 Issue #1: National Vision and Plan 68
- 7.7 Issue #2: Nature and Focus of Funding 69
- 7.8 Issue #3: Community 70
- 7.9 Issue #4: Sharing and Collaboration 71
- 7.10 Issue #5: Challenge Problems, Benchmarks, and Competitions 72
- 7.11 Skilled and Knowledgeable Personnel 73
- 7.12 Near-Term R&D Priorities 73

8 CONCLUSIONS AND RECOMMENDATIONS 77
8.1 Recommendation #1: Make a National Decision to Address the Software Understanding Gap 78
8.2 Recommendation #2: Establish a Cross-Agency Software Understanding for National Security Executive Council (SUNSEC) 79
8.3 Recommendation #3: Direct the Coordination of Software Understanding Collaboration Policies and Activities Across Agencies 80
8.4 Conclusions 80

LIST OF FIGURES

Figure 1: Scale of the Software Understanding Problem 10
Figure 2: Conceptual depiction of the accelerating Software Understanding Gap 17
Figure 3: Sources of Confidence in Mission Software in an Ideal State. 21
Figure 4: Current State vs. Ideal State Comparison in Software Understanding 23
Figure 5: Alternatives for Mission Confidence in Software in the Current State 24

LIST OF TABLES

Table 1: Abbreviations and Definitions 4
Table 2: Document Revisions 5

ACRONYMS AND DEFINITIONS

Table 1: Abbreviations and Definitions

Abbreviation	Definition
AI	Artificial Intelligence
COTS	Commercial Off-the-Shelf
CISA	Cybersecurity and Infrastructure Security Agency
DARPA	Defense Advanced Research Projects Agency
DHS	Department of Homeland Security
FFRDC	Federally Funded Research and Development Center
GOTS	Government Off-the-Shelf
ICS	Industrial Control Systems
IoT	Internet of Things
ML	Machine Learning
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
NNSA	National Nuclear Security Administration
NS&CI	National Security and Critical Infrastructure
NSA	National Security Agency
NVD	National Vulnerability Database (NIST-owned)
OUSD(R&E)	Office of the Under Secretary of Defense for Research and Engineering
R&D	Research and Development
RCE	Remote Code Execution
RE	Reverse Engineering

Abbreviation	Definition
ROP	Return-Oriented Programming
SBOM	Software Bill of Materials
SME	Subject Matter Expert
SUNS	Software Understanding for National Security
UARC	University Affiliated Research Center
USG	United States Government

DOCUMENT REVISIONS

Table 2: Document Revisions

Version	Revision Date	Revision Description
1.0	March 25, 2025	Initial public release

1 Introduction

Our nation is presently facing extensive, unmeasurable risk to our national security and critical infrastructure (NS&CI) missions through our widespread dependence on largely inscrutable third-party and legacy software. In 2011, Marc Andreessen warned that "software is eating the world."[1] In recent decades, software has been integrated into every facet of government and society, including NS&CI missions. Despite rigorous testing, software typically contains unexpected behavior that can imperil the missions that rely upon it. This problem is already expansive as the full range of software upon which missions depend is vast and the number of mission-driven questions related to that software is extensive. Yet the problem is accelerating. As software is integrated into ever more systems, it continues to grow in complexity, and various pressures have led to more rapid update cycles. The gap in our nation's ability to understand the software we depend upon is rapidly expanding. Every U.S. government agency faces the same challenge, yet there is no coordinated activity to create the necessary capabilities. The net result is that our nation faces a potentially existential threat as our vital missions and most sensitive systems rely on software components that cannot be adequately understood, leading to extensive, unknown, and unbounded mission risk. Our confidence today in NS&CI mission software is largely based on unsubstantiated assumptions, not on reliable, technical evidence. Section 2 describes this problem in more detail.

Ideally, to maximize the probability of mission success, mission owners would routinely analyze all mission-relevant software to seek technical evidence about its potential behavior before putting it into operation. After placing it into operation, owners would perform periodic analyses when new concerns come to light. Unfortunately, the technical tools and automated capabilities do not exist to do so given the broad set of mission questions, the vast array of software upon which we depend, and the speed at which answers are needed. Focused efforts to improve development practices are laudable but insufficient to address the vast volume of third-party and legacy software already in use, let alone the accelerating pace of new and updated software requiring mission analysis. Section 3 describes an ideal future state that addresses the challenges of analyzing legacy and third-party software and examines the ways in which current capabilities and approaches fall far short, creating a *software understanding gap*.

The discerning reader may wonder how it is that society and the nation arrived at a point at which critical government missions depend on inscrutable software with a demonstrated track record of failure. Section 4 traces key elements in the history of computing and government policy that have led to the software understanding gap. It explains how these policy decisions made sense at the time, but also highlights how recent trends have outdated the original assumptions upon which those decisions rested, leading to the current state of widespread, unbounded risk from third-party software. Our nation's historical choices on how to leverage software widely have resulted in economic opportunity and prosperity, but also a tremendous gap between the total dependence on software for NS&CI missions and the capability to understand and validate that software. Section 4 explains key characteristics of software that make it so challenging to reason about, including the surprising limitations of software testing and the unbounded risk we face from adversaries misusing our software systems.

---

[1] Marc Andreesen, "Why Software Is Eating the World," AH Capital Management, L.L.C. ("a16z"). (originally published in The Wall Street Journal, August 20, 2011), https://a16z.com/why-software-is-eating-the-world.

Section 5 presents several case studies and examples to illustrate the consequences of failures in the current software landscape and how unexpected behavior in software can lead to disastrous and mission-imperiling results.

There are several options for the nation to consider in how to address the software understanding gap. Section 6 discusses these options as well as technical questions surrounding these options that inform the nation's future software risk assessment.

As a first step toward confronting the technical aspects of this national challenge, five USG representatives co-convened the "Software Understanding for National Security (SUNS) 2023 Workshop" in March 2023 in Arlington, VA. Section 7 describes this workshop and summarizes results. The SUNS 2023 Workshop brought together mission stakeholders and technical subject matter experts (SMEs) from 18 government groups and Federally Funded Research and Development Centers (FFRDC) to consider the challenges and potential options. The central activity of the workshop was a three-day technical sprint during which the SMEs expressed 1) their individual opinions on the importance of the problem, the 2) feasibility of the technical capabilities that would be needed to meet NS&CI mission needs, and 3) the challenges and possible paths forward for producing them.

From the SMEs' expressed opinions, the authors of this report draw these conclusions from the workshop:

1. **A 10x-100x or more improvement in software understanding capabilities is possible:** However, progress is currently prevented by lack of a centralized vision, funding that is at least 10x too low, inability to collaborate, and other non-technical issues.
2. **Lack of A Coordinated Approach:** Although the technical capabilities needed by different agencies and missions share extensive technical commonality, the lack of coordination toward an investment-style approach results in current funding efforts squandering the potential return on investment for the broader government.
3. **The Challenge Is Rapidly Expanding:** Absent a coordinated approach to developing the needed technical capabilities the USG will continue to fall further behind in solving this rapidly expanding problem.
4. **A Unified National Effort Is Required:** A unified national effort is necessary to revolutionize our automated software understanding capabilities to meet current and future NS&CI mission needs.

Based in part on the results of the SUNS 2023 workshop, in January 2025, four U.S. government agencies released a coauthored report on "Closing the Software Understanding Gap." This report called for decisive and coordinated action across the U.S. Government regarding these issues, stating:

> *This report … is a call to action for the U.S. government to take decisive and coordinated action to close the software understanding gap. By closing the gap before other nations and obtaining a deep, scalable understanding of software-controlled systems, including artificial intelligence (AI)-based systems, the United States will secure an advantage in geopolitics for the foreseeable future and will help harden U.S. critical infrastructure from adversarial state-sponsored activity.*[2]

[2] CISA, DARPA, OUSD(R&E), and NSA, "Closing the Software Understanding Gap," January 16, 2025.

Section 7 describes the SUNS 2023 Workshop and the authors' analysis of its key findings in more detail.

Section 8 presents the authors' conclusions and recommendations for action. Appendices A-C provide additional detail on select topics. Appendix A provides examples of sample mission questions. Appendix B contains frequently asked questions (FAQ) for those less familiar with computer software and the associated issues. And finally, Appendix C contains an annotated report of the straw polls taken during the SUNS 2023 Workshop.

1.1 Acknowledgments

We would like to acknowledge the Department of Homeland Security's Science and Technology Directorate, which funded the planning and execution of the SUNS 2023 Workshop (see Section 7) as well as the creation of this document. The SUNS 2023 workshop co-conveners, listed alphabetically, are as follows:

- **Christopher Butera,** Technical Director, Cybersecurity Division, Cybersecurity and Infrastructure Security Agency (CISA)
- **Edward Jakes,** Director, Nuclear Enterprise Assurance Division, National Nuclear Safety Administration (NNSA)
- **Dr. Garfield Jones**, Associate Chief of Strategic Technology, Cybersecurity and Infrastructure Security Agency (CISA)
- **Dr. Robert Runser**, Technical Director of Research, National Security Agency (NSA)
- **Neal Ziring,** Technical Director of Cybersecurity, National Security Agency (NSA)

We would also like to thank the individual technical Subject Matter Experts (SMEs) who participated in the SUNS 2023 Workshop. The SMEs were drawn from the following organizations:

- Carnegie Mellon University, Software Engineering Institute (SEI)
- Cybersecurity and Infrastructure Security Agency (CISA)
- Defense Advanced Research Projects Agency (DARPA)
- Georgia Tech Research Institute (GTRI)
- Institute for Defense Analyses, Center for Computing Sciences (IDA/CCS)
- Lawrence Livermore National Laboratory (LLNL)
- MIT Lincoln Laboratory (MIT-LL)
- National Security Agency (NSA)
- Pacific Northwest National Laboratory (PNNL)
- Sandia National Laboratories (SNL)

Zeichner Risk Analytics (ZRA) helped design, execute, and document the SUNS 2023 Workshop. ZRA also assisted with preparing this report.

2 The Problem: Unbounded Mission Risk from Inscrutable Software

In recent decades, society and government have been transformed through the adoption of software into every aspect of life. While many systems have not changed much in outward appearance, the introduction of digital technology has changed their fundamental nature in key ways. As security researcher Bruce Schneier explains:[3]

> *Your modern refrigerator is a computer that keeps things cold. Your oven, similarly, is a computer that makes things hot. An ATM is a computer with money inside. Your car is no longer a mechanical device with some computers inside; it's a computer with four wheels and an engine.*

The same is true for systems in National Security and Critical Infrastructure (NS&CI) mission spaces. As an example, a 2018 Government Accountability Office (GAO) report on DoD weapon systems states, "Nearly all weapon system functions are enabled by computers—ranging from basic life support functions, such as maintaining stable oxygen levels in aircraft, to intercepting incoming missiles."[4] Today, most NS&CI missions now stand or fall based on the behavior of the software upon which they depend. Given this situation, on what foundation should confidence in NS&CI missions rest? In a National Research Council report, "Software for Dependable Systems: Sufficient Evidence?",[5] the authors state:

> *…a software system … should be regarded as dependable—certifiably dependable—only when adequate evidence has been marshalled in support of an argument for dependability that can be independently assessed.*

Given the momentous nature of NS&CI missions, confidence in such systems must rest on the strength of technical capabilities, with risk decisions informed by technical evidence packages of software's potential behavior, derived from forensic analysis of the software artifact itself. The nation must ensure that mission owners have the technical capabilities necessary to routinely pose any question they deem to be mission relevant about any software artifact they deem to be mission critical (this is explored in more detail in Section 3.2).

Any technical analysis of mission critical software requires two key elements: (1) a software artifact to analyze and (2) a specific question to answer. Regarding the first element, what types of software should be in view for the nation's ability to analyze? Briefly stated, *all software* upon which NS&CI missions critically depend. This includes all forms of software, including executable binaries, byte codes, intermediate forms, source code, etc. This also includes all types of systems and software, including desktop, server, mobile, embedded computing, artificial intelligence (AI) and machine learning (ML) systems, internet of things (IoT), industrial control systems (ICS), cyber physical systems, security systems, building automation, manufacturing, energy production or distribution, communication, vehicles, aviation, and many more. It further includes all layers of software used in

---

[3] Bruce Schneier, "Click Here to Kill Everyone," New York Magazine Intelligencer, January 2017, https://nymag.com/intelligencer/2017/01/the-internet-of-things-dangerous-future-bruce-schneier.html.
[4] USG, "Weapon Systems Cybersecurity," U.S. Government Accountability Office, October 9, 2018, https://www.gao.gov/products/gao-19-128.
[5] Daniel Jackson, Martyn Thomas, L.I. Millett, National Research Council, "Software for Dependable Systems: Sufficient Evidence?," National Academies Press, 2007, https://www.researchgate.net/publication/309418679_Software_for_dependable_systems_Sufficient_evidence.

these systems, whether at the cloud, application, operating system, hypervisor, firmware, boot system, or other layers.

Regarding the second element, what types of questions should be in view? There are a wide range of such questions that one could pose about mission software. A few examples include:

- *Is there an authentication bypass (e.g., a "backdoor") in this software?*
- *Under what conditions might the software turn on the camera/microphone?*
- *Does the software preserve the integrity of my critical data?*
- *Does the control system software have a remote kill switch?*

These are termed *mission questions*—high-level but specific technical questions about software and its behavior that often determine mission success or failure. For a more extensive list of examples, see Appendix A .

Answering some mission questions requires only identifying *anything of relevance* in the software under test, such as a malware mission, which can reject a software sample if anything malicious is found, or a United States Cyber Command mission in which only one exploitable vulnerability is required. Although some software behaviors of interest may be easy to detect while others may be challenging, with these types of mission questions, finding *anything* relevant is often sufficient. Other missions depend more on finding *everything* relevant to the question at hand, such as a cyber assurance mission where overlooking a single potential software behavior could threaten mission success. In these cases, the level of confidence depends on the level of thoroughness of the analysis, with complete confidence only being achieved if the analysis of the software is complete.

Figure 1, shown below, illustrates that numerous agencies have similar but extensive questions regarding a vast array of software.

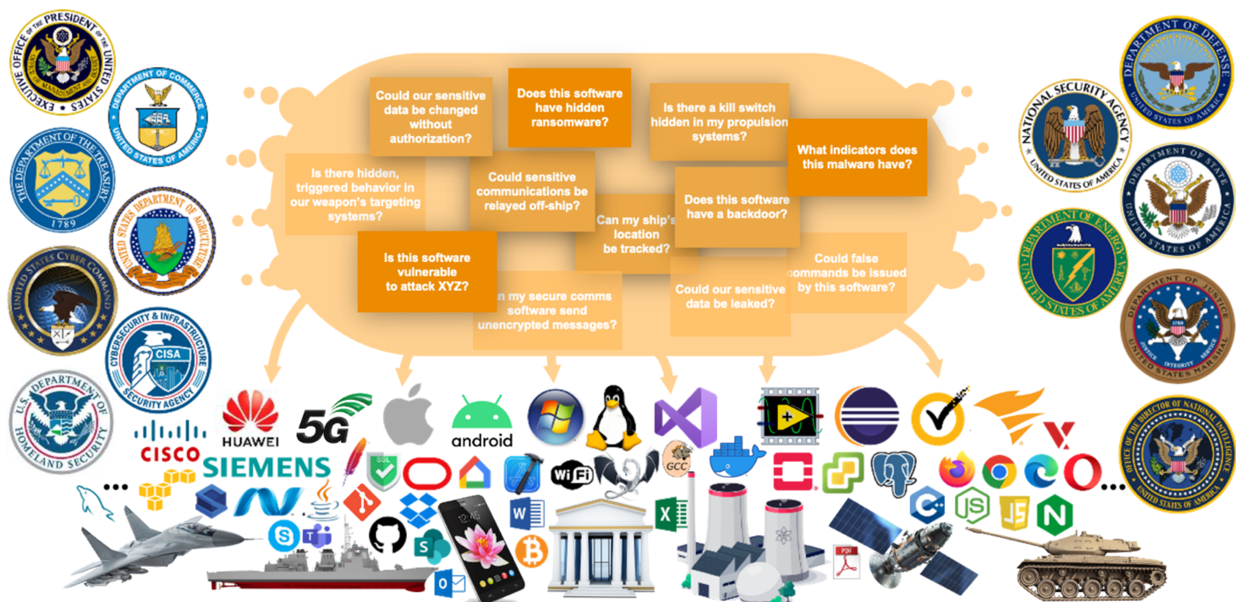


Figure 1: Scale of the Software Understanding Problem

Unfortunately, answering most mission questions about the software systems that underpin government and society is not straightforward. As the next section discusses, the software that has

transformed our NS&CI mission spaces has a significant negative downside that has lead to unbounded mission risk, as discussed below.

2.1 Unexpected Behavior in Software

Most software is replete with latent, unexpected behavior, despite the care that goes into creating it and the testing that goes into ensuring it works correctly. One example of this comes from the Therac-25, the first software-controlled, medical radiation therapy machine.

2.1.1 Therac-25

From 1985 through 1987, the Therac-25 was involved in massive radiation overdoses of at least six patients, resulting in severe injury and death.[6] This unexpected behavior was unintentional. It remained present but hidden throughout the design, the development, the testing, the safety analysis, and the successful and safe treatment of many patients. When very specific conditions arose, the unexpected behavior emerged and produced a wildly incorrect dose of radiation. The observed behavior of the system baffled the developers of the Therac-25, even after a careful review of the software's design and source code.

A 1986 letter from the technical support supervisor of the Therac-25 stated, "After careful consideration, we are of the opinion that this damage could not have been produced by any malfunction of the Therac-25 or by any operator error." In short, the developers *believed the observed behavior to be impossible*, but it was possible. The system would go on to take more lives before the developers finally understood the subtle ways in which their software was misbehaving and putting patients in danger. The investigation report concludes, "Virtually all complex software will behave in an unexpected or undesired fashion under some conditions... ."

Despite the best intentions and concerted efforts of the Therac-25 developers, the software controlling the system had unexpected behavior that altered the performance of the radiation machine dramatically—even fatally—at times. When the system developers were faced with concrete evidence of the system's fatal behavior, they did not understand how it could be possible. If the "mission" of the radiation therapy machine is to always deliver the proper dose, then this unexpected behavior in the software caused mission failure.

2.1.2 Implications for National Security & Critical Infrastructure Missions

There are two valuable lessons we can draw from this example that apply to the use of software in NS&CI missions. First, developer attestation must not be the cornerstone of confidence in vital software systems. Regardless of the skill of software developers, the nature of software systems give rise to subtle, unexpected behavior that defies human ability to predict (see Section 4). This example illustrates that risk decisions about high consequence systems require a set of technical capabilities designed to reason about such potential behavior—a set of technical capabilities which the USG is not currently on track to develop.

Second, many people, including experts, misunderstand the nature of testing, thinking that it means more than it does and drawing conclusions of high confidence that are in fact unwarranted. While testing can *validate behavior that is expected to be present*, except in a few special cases, it is not capable of *ruling out unexpected behavior*, such as the latent unexpected behavior present in the

[6] Nancy Leveson and Clark Turner, "An Investigation of the Therac-25 Accidents," IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41, https://ieeexplore.ieee.org/document/274940.

Therac-25 that evaded rigorous testing (for a deeper discussion, see Section 4.3.1). Thus, in NS&CI missions, high confidence in the mission requires a technical basis that extends beyond testing.[7]

2.1.3 Prevalence of Unexpected Behavior

Unfortunately, unexpected behavior in software is far from rare. The National Institute of Standards and Technology (NIST) maintains the National Vulnerability Database (NVD), which records reported software flaws. In 2022, there were over 25,000 vulnerabilities reported with an average year-over-year growth of over 15% for the past 20 years.[8] A vulnerability is one type of unexpected behavior. Each of these represents an instance in which the software in question exhibited behavior that was likely unexpected by the original developers and put the system at potential risk. It is critical to realize that the NVD entries represent behaviors that were discovered *after the software was already put into use*.

2.1.4 Sources and Consequence of Unexpected Behavior

Software vulnerabilities are the most widely recognized type of unexpected behavior in software. Vulnerabilities arise from a variety of root causes and can have a wide range of effects, including system crashes, incorrect system decisions, information loss, unexpected data manipulation or deletion (as occurs in ransomware attacks).[9] The most alarming effect is known as *remote code execution (RCE)* in which an attacker asserts positive control over a system's operation, which may be done covertly or overtly.

Vulnerabilities arise when a rule or expectation is violated. At the lowest level, vulnerabilities arise from violating a rule of math (e.g., division by zero), computing (e.g., using a hardware register that has not been initialized), or software programming (e.g., using memory that has not been allocated). These types of vulnerabilities are widely understood[10,11,12] and many can be addressed with universal solutions or mitigations. As a result, there are concerted, ongoing efforts to engineer technical controls into languages, compilers, and processors to prevent these classes of vulnerabilities and to encourage the widespread adoption of these solutions.[13,14] More concerning are high-level vulnerabilities that arise by violating application-specific or organization-specific expectations or rules. Unlike low-level vulnerabilities, these vulnerabilities do not have a limited number of well-understood classes with universal solutions or mitigations. In the limit, each is bespoke, requiring domain-specific knowledge to understand and successfully mitigate. The consequences, likewise, of such high-level vulnerabilities are wide ranging and defy categorization.

In many cases, there is disagreement on whether a particular behavior is a vulnerability or feature. Consider the ability to remote track a mobile device's location without user action—this may be

---

[7] The one exception to this is if the testing is exhaustive and covers all possible states in the system. In these cases alone can testing give assurance that undesirable behavior is absent from a system.

[8] National Institute of Standards and Technology (NIST), "NIST National Vulnerability Database," Accessed Oct 29, 2023, https://nvd.nist.gov/.

[9] CISA, "Ransomware 101," Accessed Oct 29, 2023, https://www.cisa.gov/stopransomware/ransomware-101.

[10] MITRE, "CWE Top 25 Most Dangerous Software Weaknesses," Accessed Jul 12, 2024, https://cwe.mitre.org/top25/.

[11] Dowd, McDonald, and Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*, (Addison-Wesley Professional, 2006).

[12] MITRE, "2021 CWE Most Important Hardware Weaknesses," Accessed Oct 29, 2023, https://cwe.mitre.org/scoring/lists/2021_CWE_MIHW.html.

[13] Office of the National Cyber Director, "Back to the Building Blocks: A Path Toward Secure and Measurable Software," February 2024.

[14] Cybersecurity & Critical Infrastructure Security Agency, "Secure by Design," Accessed July 12, 2024, https://www.cisa.gov/securebydesign.

considered a feature to a company tracking its fleet of vehicles but may be considered a vulnerability in a personal security scenario. Remote, off-site administrative access may be a valuable feature to a company intending to use it for continuity of operations, or a vulnerability if the company was unaware of it and would have blocked it had they been aware of it.

There are many types of unexpected behavior that threaten NS&CI missions and that go well beyond the traditional definitions of software vulnerabilities. The Volkswagen case study (see Section 5.1) is one such example that is a mission-threatening behavior that would not normally be categorized as a vulnerability. Many other examples can be found in the extended discussion of mission questions in Appendix A .

While capabilities to look for low-level and known categories of vulnerabilities are important, establishing confidence in the software used in NS&CI missions requires more than just that. Mission owners must also have the latitude to pose a wide array of mission-specific questions of the software upon which their missions depend.

There is a malicious twist to this problem. Given our national inability to detect unexpected behavior, if an adversary were to insert malicious behavior into software used in NS&CI missions, how would it be detected? GitHub's 2020 Octoverse report, "Securing the world's software", reported that 17% of vulnerabilities surveyed were maliciously inserted.[15] The SolarWinds Orion (see Section 5.2) and Node-ipc incidents (see Section 5.4) stand as cautionary tales for which there is currently no adequate answer that scales to the full set of relevant software systems and mission questions in view.

2.2 Lack of Adequate Solutions

In NS&CI missions, it should be considered unacceptable to place software into service in critical applications that is later determined to have unexpected behavior. There are a variety of currently available approaches that might be considered for analyzing legacy and third-party software to discover unexpected behavior before deeming it safe to place into use, but none are sufficient to solve this problem.

One of the most common approaches is testing. As illustrated in the Therac-25 example above, testing is not capable of leading to high confidence that unexpected behavior is absent from software. According to the National Research Council:[16]

> *The theoretical inadequacies of testing are well known. To test a program exhaustively would involve testing all possible inputs in all possible combinations and, if the program maintains any data from previous executions, all possible sequences of tests. This is clearly not feasible for most programs, and since software lacks the continuity of physical systems that allow inferences to be drawn from one sample execution about neighboring points, testing says little or nothing about the cases that were not exercised.*

Vulnerability scanning tools are available, but only a small subset of the mission questions of interest falls into the low-level, universal vulnerability types that these scanners are designed to detect. Malware scanning tools are designed to check for patterns that are known *a priori* to be malicious but are based on approaches that are not well suited to detecting novel malicious

[15] GitHub, "The 2020 State of the Octoverse," Accessed Oct 29, 2023, https://octoverse.github.com/2020/.
[16] National Research Council, "Software for Dependable Systems: Sufficient Evidence?" (National Academies Press, 2007).

behavior and that exercise design tradeoffs better suited to consumer markets than to NS&CI needs. Answering the full range of mission-specific questions at a level of rigor necessary for NS&CI missions requires fundamentally different approaches.

Manual reverse engineering can successfully analyze a small selection of key software for technical evidence to answer mission questions, however, this pool of talent is limited. Further, the manual approaches already cannot keep up with present NS&CI mission needs and cannot scale to answer the breadth of questions about the vast array of mission-relevant software at the pace needed. Section 3.3 discusses current approaches in more detail.

Recent efforts to address the problem—through more rigorous development practices, secure-by-design best practices, requirements for a software bill of materials (SBOM), vendor attestation, and even mathematically rigorous approaches to designing software—are all laudable. However, these efforts fall short in answering the NS&CI need to analyze software for undesirable behavior:

1. **3rd Party and Legacy Software:** These efforts do not address the vast volume of third-party and legacy software already in use.
2. **Questions Unanticipated at Design:** Even if software could be built without any flaws, NS&CI missions depend on a large volume of software not designed explicitly for them; in such cases, NS&CI mission owners need answers to mission-specific questions that the original designers may never have anticipated.
3. **Independent Post-Construction Analysis:** Even once radically improved development practices are adopted, the SolarWinds Orion incident mentioned above together with the National Research Council's call for "independent assessment" demonstrate the need for independent technical analysis of software program executables post-construction.

In summary, existing tools and approaches fall far short of NS&CI mission needs for rigorous, reliable, repeatable, and rapid analysis of mission-critical software for unexpected behavior. The next section describes current trends in software adoption and production that reveal that the situation is growing rapidly worse.

2.3 An Accelerating Problem

This section describes three trends that are increasingly exacerbating the national challenge of understanding software sufficiently to ensure mission success: 1) expanding adoption, 2) increasing complexity, and 3) faster updates.

Expanding Adoption. The expanding adoption of software across all major segments of government, society, and the economy continues to broaden the nation's reliance on it. Today, software is already ubiquitous, having permeated the power grid, telecommunications, transportation, government services, medical services, banking systems, border control, military systems, and other segments of national security and critical infrastructure. Beyond this, private industry and government are leveraging the transformative power of software to usher in smart agriculture, smart buildings, and smart manufacturing. Propelled by 5G innovations and plans for ubiquitous connectivity, software is promising to usher in smart cities by using automation and AI to manage our traffic, parking, energy production and distribution, mass transit, waste and recycling, and first responder services. Calls can already be heard for smart military bases[17] and smart

[17] Erv Lessel, Bill Beyer, Ted Johnson, "Smart Military Bases," Deloitte Insights, March 30, 2017, Accessed Dec 2, 2023, https://www2.deloitte.com/us/en/insights/industry/dcom/byting-the-bullet-smart-military-bases.html.

government.[18] The rapidly expanding number of systems that depend on software in NS&CI mission spaces places an accelerating urgency on our need to adequately understand software's potential, mission-threatening behavior.

Increasing Complexity. At the same time, software continues to grow in complexity. Decreases in the cost, size, and power requirements of computing hardware, coupled with increasing compute, memory, and storage capacity, mean that computers are shrinking while growing more powerful. These trends produce more capacity for software companies to step in with ever more functionality and features. As the difficulty of analyzing software grows alongside this increase in software complexity, existing stopgap measures such as manual reverse engineering become rapidly obsolete, demanding a new generation of automated techniques.

Faster Creation and Updates. Software is being created faster than ever. Driven in large part by the growing number of vulnerabilities—an important type of the unexpected potential behavior addressed in this report—software vendors are releasing patches, updates, and new versions at rates inconceivable in past decades. Ubiquitous internet and Wi-Fi connectivity in many segments of government and society make these updates easier than ever to install and is fueling the trend toward silent background updates of which the user is typically unaware. In recent years, software vendors have introduced AI programming assistants into the software development lifecycle, enabling developers to produce software dramatically faster. Unfortunately, studies find that, all too often, the code generated has vulnerabilities.[19]

As software is increasingly integrated into more and more NS&CI systems, as software becomes increasingly complex, and as it is created and updated more rapidly, the pace at which mission questions will need to be answered about it will continue to accelerate. Mission owners are already in an environment in which they cannot answer most mission questions about software at the speed necessary to meet mission requirements.

As an illustration of this point, consider this redacted, real-world example request from a mission owner:

> *In support of the <redacted, mission-critical> project, I need a risk assessment of <multiple, third-party, interconnected, enterprise software applications>. All of these software components will be on internet-connected machines. The desire is to ensure that this software is free from vulnerabilities, malware, intentionally added "backdoors" and security flaws before it is installed. This request is urgent as we are ready to use the software immediately but must have it scanned before installation.*

The deadline that the mission owner submitted with this request was less than one week from submission. This mission owner is to be applauded for submitting such a request, as this is what the mission requires. This mission owner, however, did not realize that the gulf between the reasonable request and the technically feasible is vast. A complete technical assessment of this scope typically requires more than a year of effort from several expert reverse engineers and/or system security experts, and even then, would not come close to the guarantee of freedom from vulnerabilities this

[18] Rana Sen, Miguel Eiras Antunes, Mahesh Kelkar, "Smart Government," Deloitte Insights, June 24, 2019, Accessed Dec 2, 2023, https://www2.deloitte.com/us/en/insights/industry/public-sector/government-trends/2020/data-driven-government.html.

[19] Vahid Majdinasab, et al., "Assessing the Security of GitHub Copilot's Generated Code—A Targeted Replication Study," PROMISE 2022, Singapore, Nov 18, 2023, Accessed Dec 2, 2023, https://dl.acm.org/doi/pdf/10.1145/3558489.3559072.

mission owner needs. This serves as a poignant illustration of an important, reasonable request that is currently impossible to fulfill at acceptable timeframes and cost.

For many NS&CI missions, even a delay of one week to analyze a software package to answer a mission question can be unacceptable. When a NS&CI mission discovers an actively exploited vulnerability in critical software, not only is minimal delay in understanding and fixing the problem paramount, but so is a high degree of confidence that the fix does not also inadvertently introduce unexpected new behaviors that may imperil the mission. Mission owners need answers to their questions about software at the pace of mission.

2.4 Conclusions Regarding the Software Understanding Problem

The government and society trust much software that they shouldn't. Much of this trust is without a technical basis derived from the software itself and is therefore unwarranted. The Therac-25 incident illustrates the fact that unexpected, mission-threatening behavior can persist deeply hidden in software despite testing and attestations by those who developed it. This behavior can be present even when the designers, upon scrutiny, declare it to be impossible based on their design intent, development practices, and testing.

For many years now, it has been known that software can fail in unexpected ways,[20] yet such failures are disturbingly and increasingly common in present day systems. For critical systems, such as in NS&CI mission spaces, the discovery of unexpected behavior in software is evidence that, ideally, *that software should never have been placed into service in the mission in the first place*. Unfortunately, the lack of capabilities necessary to reason about software to detect mission-imperiling behaviors in advance leaves mission owners with no effective alternative. With no alternative, mission owners are blindly accepting an unknown amount of risk and placing software into operation without an adequate technical evidence package characterizing its potential behavior.

The scope of NS&CI mission dependence on software is vast. The questions that should be routinely asked about it are equally vast. The technical capabilities required to meet this need for existing legacy and third-party software do not exist. At the same time, this already dire situation continues to deteriorate as rates of software adoption, complexity, and production accelerate, while vulnerabilities and other unexpected behaviors rise. Consequently, our nation is presently facing extensive, unmeasurable risk to our NS&CI missions through our widespread dependence on largely inscrutable third-party and legacy software.

Every U.S. government agency faces this same challenge, yet no coordinated activity is creating the necessary capabilities.

In writing about the Challenger shuttle disaster and its root causes, Diane Vaughan defined the term "normalization of deviance" as, "the gradual process through which unacceptable practice or standards become acceptable. As the deviant behavior is repeated without catastrophic results, it becomes the social norm for the organization."[21] In 2023 the director of the Cybersecurity and Infrastructure Security Agency (CISA), Jen Easterly, applied this concept to modern computing systems and software in this statement:[22]

---

[20] Robert Charette, "Why Software Fails," IEEE Spectrum, September 2005, https://spectrum.ieee.org/why-software-fails.
[21] Diane Vaughan, "The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA" (University of Chicago Press, 1997).
[22] Jen Easterly, "Unsafe at Any CPU Speed: The Designed-in Dangers of Technology and What We Can Do About It," remarks to Carnegie Mellon University, February 2023, https://www.cisa.gov/cisa-director-easterly-remarks-carnegie-mellon-university.

We've normalized the fact that technology products are released to market with dozens, hundreds, or thousands of defects, when such poor construction would be unacceptable in any other critical field.

This normalization of deviance and sustained lack of adequate capability for years has produced a state of learned helplessness, in which government leaders and NS&CI mission owners have learned to down-scope expectations and the mission questions posed to match the capabilities at hand. This is understandable but should not be acceptable in NS&CI mission spaces in which mission failure may lead to national-scale catastrophe.

The net result is that society faces a potentially existential threat as its vital missions and most sensitive systems rely on software components that cannot be adequately characterized, leaving mission owners with no recourse but to accept extensive, unknown, and unbounded mission risk through use of inscrutable legacy and supply chain software. Few fully appreciate the extent to which our confidence today in NS&CI mission software is largely based on unwarranted and unsubstantiated assumptions.

3 The Software Understanding Gap

The threat described above stems from the *software understanding gap*—a gap that exists because our ability to build software greatly outstrips our ability to understand it. This section defines software understanding and characterizes the gap in software understanding by contrasting an ideal state of managing software risk with the current state, then draws conclusions from the comparison. Section 4 will examine the history and characteristics of software that have led to this gap.

As shown conceptually below in Figure 2, the ability to produce software has outstripped our ability to adequately understand it. The gray space between the two is *the software understanding gap*.

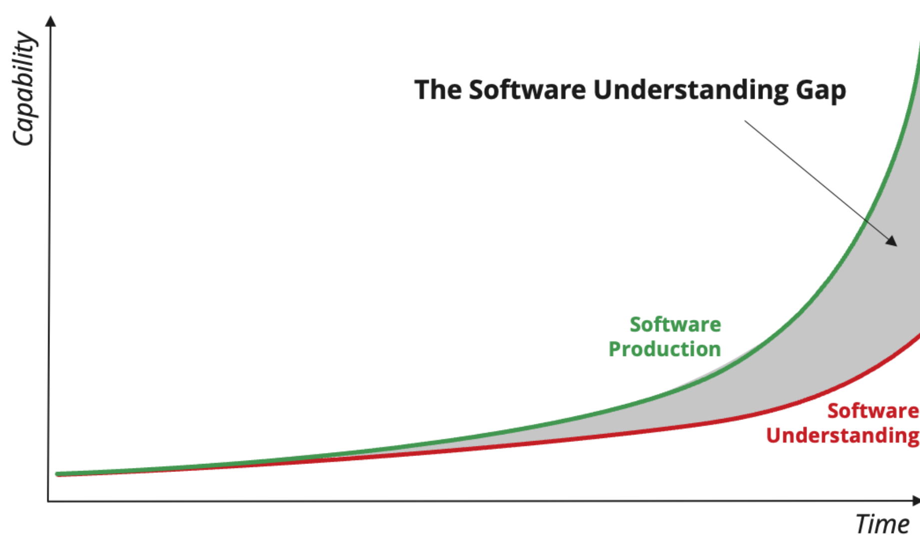


Figure 2: Conceptual depiction of the accelerating Software Understanding Gap

3.1 Software Understanding Defined

As used in the "Closing the Software Understanding Gap" report[2], *software understanding* is defined as: *the practice of constructing and assessing software-controlled systems to verify their functionality, safety, and security across all conditions (normal, abnormal, and hostile)*.

This report uses a more detailed definition of the term *software understanding* as follows: *the socio-cognitive practice of discovering the behavior (both observed and possible) of software by analyzing the software artifact itself (either in isolation or within a system) sufficiently to answer a given mission question, rather than relying on proxies such as documentation, developer attestation, or development processes.* The term "socio-cognitive" indicates the end goal is for people, including mission owners, to understand the potential behavior and communicate it to others as part of a collective decision about the benefits, risk, and mitigation options for using the software. The term "software artifact," indicates the diverse forms and types of software listed in Section 2.

The term *software understanding* does not mean that *everything* about a particular software sample is known, but that *enough* is known relative to a particular mission question to gain sufficient technical evidence of the software's potential behavior for a mission owner to make an informed, evidence-based risk decision.

3.2 The Ideal State of Rigorous, Reliable, Rapid, and Repeatable Software Understanding

Recent Executive Orders (EO), national-level strategies, and policy directives have taken steps to address national concerns about our use of software. EO 14028[23] advocates for new architectures such as Zero Trust, which promotes the use of tools to check for known and potential vulnerabilities and SBOMs. Due to the "extraordinary potential for promise and peril" of AI, EO 14110 aims to improve the nation's ability to understand and mitigate risks from the use of AI software.[24] Various other USG memoranda and strategies seek to ensure software products function properly,[25] enhance the software supply chain,[26] and shape market forces to drive security and resilience.[27]

EO 14144, "Strengthening and Promoting Innovation in the Nation's Cybersecurity", includes many elements related to software understanding.[28] The EO puts forth requirements for strengthening security, improving accountability for software and cloud service providers, and promoting innovation, including the use of emerging technologies. Improving accountability across software and cloud service providers goes beyond software security and operationalizes the main premise of the software understanding concept.

---

[23] USG, "Improving the Nation's Cybersecurity," Executive Order 14028, May 12, 2021, https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[24] USG, "Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence," Executive Order 14110, October 30, 2023, https://www.whitehouse.gov/briefing-room/presidential-actions/2023/10/30/executive-order-on-the-safe-secure-and-trustworthy-development-and-use-of-artificial-intelligence/.

[25] USG, "Protecting Critical Software Through Enhanced Security Measures," M-21-30, August 10, 2021, https://whitehouse.gov/wp-content/uploads/2021/08/M-21-30.pdf.

[26] USG, "Enhancing the Security of the Software Supply Chain through Secure Software Development Practices," M-22-18, September 14, 2022, https://www.whitehouse.gov/wp-content/uploads/2022/09/M-22-18.pdf.

[27] USG, "2023 National Cybersecurity Strategy," March 2023, https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf.

[28] United States, Executive Office of the President. Executive Order #14144: Improving the Nation's Cybersecurity. January 16, 2025. *Federal Register.* https://www.federalregister.gov/documents/2025/01/17/2025-01470/strengthening-and-promoting-innovation-in-the-nations-cybersecurity.

Most recently, Software understanding has shown to be critical in modernizing software acquisition at the Department of Defense (DoD). As directed in a Memorandum for Senior Pentagon Leadership, the DoD is reassessing its software acquisition process. The DoD highlights that, "While commercial industry has rapidly adjusted to a software defined product reality, DoD has struggled to reframe our acquisition process from a hardware-centric to a software-centric approach."[29].

While these efforts are laudable, they fall short of achieving the ideal state for mission owners. In the ideal state, mission owners would have the ability to routinely ask any mission-related question of all mission-relevant software and receive rigorous, reliable, rapid, and repeatable answers.

- **Rigorous** answers address not only easily testable behavior, but also potential behavior that would elude traditional testing. Rigorous answers also provide a technical evidence package that links back to the specific elements of the software itself.
- **Reliable** answers are not subject to deceitful practices, stemming either from self-seeking motivations on the part of the developer, or malicious actions on the part of a hostile adversary.
- **Rapid** answers can be produced at a speed sufficient to meet mission needs, even if that speed is hours or minutes.
- **Repeatable** answers are not derived from variable processes that may give different answers at different times, as happens with intuition-based approaches.

As will be demonstrated in the case studies of Section 5 below, examining source code is often insufficient for a rigorous analysis. The source code represents an intention of what the final software executable should do, but a variety of factors can introduce differences between intentions of the developers and actual behavior of the software as it runs. The most rigorous and reliable answers to mission questions will be based on an independent analysis of the software executable that will run on the mission system.

No one knows the extent to which the ideal state is feasible or exactly what measures must be taken to reach it. However, examining the ideal state through a thought experiment can reveal how our current, real-world capabilities are falling far short of the ideal.

Due to the fundamental characteristics of software (discussed in Section 4 below), even in the ideal state, not all mission questions or software under test can be analyzed in a fully automated fashion with perfect precision: automated perfection is mathematically impossible. Important cases would still need software reverse engineers to configure tools, superintend processing, or complete challenging analyses. Thus, this ideal state is about *monumental progress*, not *perfection*.

Even partial success can be transformational. To the extent that automated software analysis tools can deliver the types of answers described above, missions will benefit as these capabilities help prevent mission-imperiling situations arising from unexpected behavior. Automation could also reduce the time required by the limited pool of highly trained software reverse engineers to provide these answers and could also enable the engineers to produce answers with higher rigor and lower cost. Whenever, however, and to what extent this ideal state of rigorous, reliable, rapid, and repeatable software understanding can be realized, NS&CI missions would have much higher confidence in their use of third-party software.

[29] Memorandum for Senior Pentagon Leadership Commanders of Combatant Commands Defense Agency and DOD Can Field Activity Directors, https://media.defense.gov/2025/Mar/07/2003662943/-1/-1/1/DIRECTING-MODERN-SOFTWARE-ACQUISITION-TO-MAXIMIZE-LETHALITY.PDF.

Both users and producers of software would benefit from this ideal state in which rigorous, reliable, rapid, and repeatable software understanding is readily achievable.

3.2.1 Software Users

In this ideal state, mission owners could routinely ask and answer all relevant questions of all mission-relevant software. Questions asked could relate to universal concerns about software, such as common vulnerabilities, as well as precise questions relating to the specific mission, operational scenarios, risk exposure, and current threat environment. The foundation of trust between the user and producer would be firmly rooted in the notion of independent analysis. In NS&CI missions, sensitive mission particulars need not be shared with the vendor when the user possesses the capabilities to perform an independent technical analysis.

If mission owners were able to receive rigorous, reliable, rapid, and repeatable answers to their mission questions about software before putting that software into use, they would have a variety of options for what to do with a given software article. If there are no adverse answers to their mission question, they could have higher confidence in using that software in the mission—a confidence based on technical analysis. Alternatively, if they discover unexpected behaviors that negatively affect the mission, but which, once known, can be mitigated with confidence, then mission owners could make a wholistic, technically informed risk acceptance decision while maintaining mission confidence. Further, if they discover egregious potential behaviors in the software, then mission owners could reject the software *before placing it into service* in particular mission systems, even if the particulars permit its use in less sensitive or less exposed areas within the organization. If mission owners could evaluate different software configurations for their impact on the potential behavior of software, they could make informed tradeoff decisions regarding which configuration parameters raise and lower mission performance vs. risk. Such an ability to rigorously understand software in advance of use could inform acquisition decisions, empowering mission owners with valuable insight that could reduce the frequency of costly scenarios in which the mission becomes dependent on software that they later discover has mission-imperiling behavior.

Different types of missions would benefit from this ideal state. As discussed more in Appendix A , many missions merely leverage software to ensure that their own mission systems function, while other missions focus specifically on analyzing software as a key aspect of the mission. Within NS&CI missions, not all systems warrant the same levels of rigor. In this ideal world, mission owners would have practical tradeoff options, making mission-specific decisions among rigor, speed, precision, and cost in terms of resources. In one scenario, a mission owner may select software in a key system for in-depth analysis, applying significant resources and opting for a lengthy analysis. In another scenario, in which a specific threat scenario is discovered, software in all systems could be rapidly analyzed for susceptibility to the threat.

3.2.2 Producers of Software

In this ideal state, software producers would routinely inspect their own software. In the case of the Orion software platform from SolarWinds, for example, the company's reputation was materially damaged when it unknowingly distributed a malicious backdoor added to their software executable by a foreign intelligence service. Unknowingly distributing malware within a company's software has occurred on many other occasions with other companies as well.[30] In this ideal state, those producers desiring to maintain a market reputation for trustworthy software would be able to apply

[30] CISA, "Defending Against Software Supply Chain Attacks," April 2021, https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf.

rigorous, reliable, rapid, and repeatable analyses to their own final software executables before shipping to market. Figure 3 depicts a notional illustration of ideal state software production—even if rigorous software development techniques become pervasive, software developers are also software users. In the ideal world, software developers would be able to adequately analyze the third-party software they have integrated into their products.

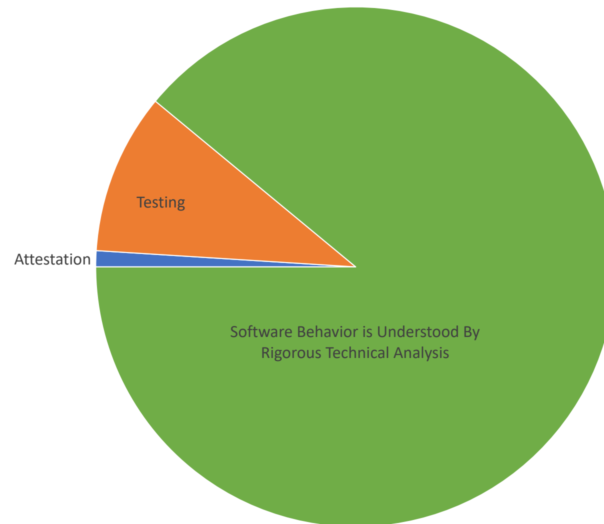


Figure 3: Sources of Confidence in Mission Software in an Ideal State.

Software producers are also software consumers. Much of the software produced today incorporates many third-party commercial and open-source libraries within it. Unexpected behavior within such third-party software can lead to unexpected behavior in the software using it, which in turn can lead to crashes, data loss, privacy loss, and other undesirable behaviors that software producers want to avoid. In the case study of the Log4j library (see Section 5.3) a feature added to a widely popular library led to a highest-possible-ranked critical vulnerability in the approximately 200,000 applications using it. In this ideal state, producers would be able to ask and answer mission questions about the software upon which their companies and their customers critically depend.

3.2.3 Analysis Tools in the Ideal State

In this ideal state, manual reverse engineering would cease to be the principal driver of deep analysis of key software. Rather, automated capabilities would answer the vast number of mission questions of the numerous systems relevant to NS&CI missions at the speed needed by the missions. Recent decades have seen the explosion of a rich, extensive, growing ecosystem of software development tools and libraries with increasing levels of automation. There has been no comparable explosion in software analysis tools, creating an accelerating gap in software understanding (see Figure 2).

In the ideal world, a similarly rich, extensive, and growing ecosystem of components, libraries, frameworks, and tools would be available to produce analysis tools customized to specific mission questions and software systems under test. The ecosystem would scale to cover the full range of mission questions that mission owners have (such as those listed in Appendix A) and operate over the broad array of NS&CI systems (such as those listed in Section 2). The resulting tools would

produce technical evidence packages that would enable software users and producers to understand potential behavior of software and inform mitigations and risk decisions.

Further, to address the scale of the challenge discussed in Section 2, a growing ecosystem of analysis components, libraries, frameworks, and tools would be developed using an investment strategy, identifying widely reusable components, designing standardized interfaces and data exchange formats, creating interchangeable parts with varying characteristics for a variety of uses, etc. Such an ecosystem would negate the need for individual mission owners to create bespoke point solutions that have little future dividend for other mission questions, other systems under test, and other agencies. All agencies and partners would contribute to solving the common problem of software understanding. Sensitive NS&CI missions would likely restrict some key software components from being shared, but those missions could still construct their analysis tools using the widely shared and reusable ecosystem.

Although automation is a necessary element in this ideal world, software providers must develop this ecosystem of analysis components, libraries, frameworks, and tools with humans in mind. Technical evidence packages must be understandable to high-level decision makers and mission owners who may not have expertise in technical software analysis. At the same time, highly trained specialists may need to be able to rapidly assemble analysis tools from the ecosystem and to superintend the analysis process, stepping in when automation fails. The long-term goal of this future ecosystem must be to increase the opportunities for all individuals to contribute to and benefit from software understanding solutions, just as the current ecosystem of software development components has enabled an ever-broadening pool of individuals to rapidly produce new software.

Figure 4 represents a notional comparison between the current state and ideal future state in software understanding. In the current state individual efforts are heavily bespoke and manual, with inadequate focus on interoperability, interchangeability, standardization, and reusability; in the ideal future state, each of these underexplored characteristics have been brought together to produce a dramatic increase in automation.

3.2.4 Society

In past decades, government investments in NS&CI missions have "trickled out" into society and created extensive benefit. In this ideal world of software understanding, society would similarly reap staggering benefits from the ability to rigorously, reliably, rapidly, and repeatably analyze software to answer a wide range of questions. A 2022 report from the Consortium for Information & Software Quality estimated the cost to U.S. society arising from poor software quality to be $2.4 trillion per year;[31] another 2022 report estimated the worldwide cost of cybercrime to be $7 trillion.[32] Unexpected and undesirable behavior in software drive both these costs.

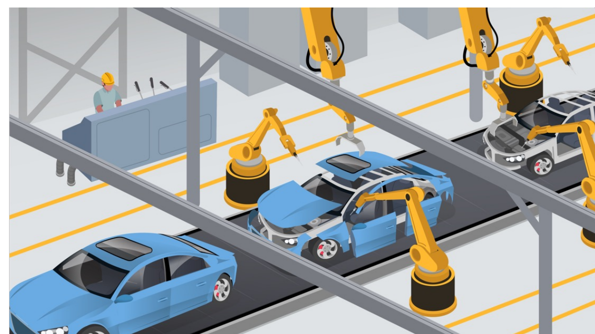
In the ideal world, software developers would analyze their software for a wide range of undesirable potential behavior before bringing it to market, thereby reducing software-related data breaches, identity theft, and financial losses. Individual privacy would increase as it relates to software use. Adversaries would find it more difficult to achieve their goals because it would be more challenging to find situations where software consumers or producers placed their trust in untrustworthy

---

[31] Herb Krasner, "The Cost of Poor Software Quality in the US: A 2022 Report," Consortium for Information & Software Quality, December 15, 2022, https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf.
[32] Steve Morgan, "Boardroom Cybersecurity 2022 Report," Cybercrime Magazine, August 10, 2022, https://cybersecurityventures.com/boardroom-cybersecurity-report/.

software. The number of cyber incidents arising from software vulnerabilities would decline. New segments of society looking to the transformational power of software, including smart agriculture, buildings, and manufacturing, could all do so with increased confidence and decreased risk to society.

(a) Notional illustration of the current state.



(a) Notional illustration of the ideal future state.

Figure 4: Current State vs. Ideal State Comparison in Software Understanding

In consumer markets, independent analysis would also provide benefit to the public. The software industry could benefit from an independent evaluation for software analogous to the “crash test dummies” of the automobile industry. The “crash test dummy” tests used by the National Highway Traffic Safety Administration (NHTSA) and other automotive testing organizations provide independent sources of information to guide consumer choices in automobile purchases. Although such testing for software cannot give equivalent confidence about software’s potential performance (see Section 4.3.1), in this ideal state, the public could base their software choices on the strength of technical analysis of the software itself, rather than on indirect, secondary indicators, such as past performance, market share, or certifications of the company’s intentions, policies, or development practices. Independent consumer advocacy organizations could provide reports of unexpected behavior in software potentially affecting privacy, security, reliability, and more, promoting market competition. Although not all software is equally amenable to automated analysis, this alone could be a relevant acquisition criterion—certain consumers may prefer software that is analyzable to software that remains inscrutable to analysis.

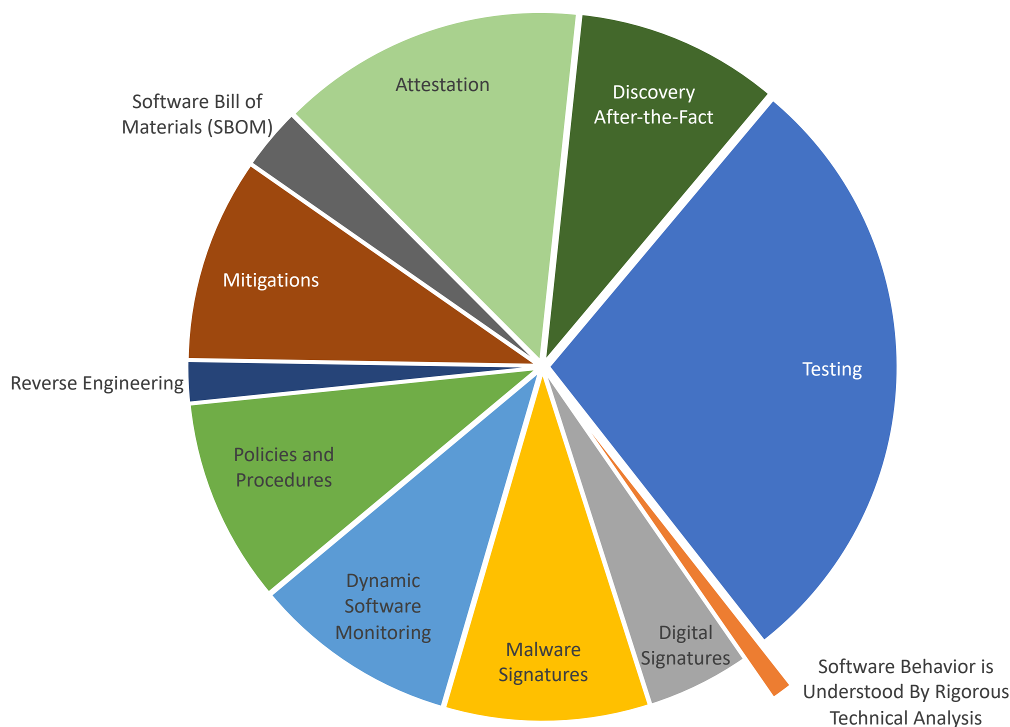


Figure 5: Alternatives for Mission Confidence in Software in the Current State

3.3 The Current State of Software Understanding Alternatives

Unfortunately, the current state in software understanding is radically different from the ideal one described above (Section 4 will explore why this is the case). This section examines the current alternatives to robust software understanding and the ramifications of relying on them relative to the capabilities envisioned above in the ideal state. Most of these alternatives have merit; however, as will be seen, from the perspective of understanding potential behavior of software adequately to answer mission questions with confidence, these alternatives are often weak and leave a significant understanding gap.

Figure 5 is a notional illustration of the current reliance on software understanding alternatives in NS&CI missions, in the estimation of the authors. It illustrates the predominance of non-rigorous proxy and alternative approaches over rigorous behavioral analysis of the software itself.

The subsections below discuss several of these.

3.3.1 Testing

One of the most popular ways to answer questions about software is to test it.

Testing is a dynamic technique that involves running the software with a variety of different inputs and observing the resulting behavior or output for each. It is used to verify that the features or behavior that are expected to be present are indeed present. Testing can also be used to check whether specific inputs result in failure, such as a software crash.

Fuzzing is a type of testing that generates random or pseudo-random inputs to check for such failures.[33] One common approach to testing is to run the software in an artificial test environment, a *sandbox*, to isolate any misbehaviors in the software from operational systems. In all these approaches, each test uses a specific input to exercise one set of *states* of the software—that is, one sequence of behavior driven by the inputs. Due to its complexity, nearly all software has more possible states than there are atoms in the universe (see Section 4.2.4 for further discussion). This is a well-known problem, commonly referred to as the *state explosion problem*.[34] Consequently, in all but a few extreme cases, it is not possible to test all possible states.[35] It is notable that, given the vast possible set of states, some experts find it surprising that testing works as well as it does. Turing Award winner Tony Hoare comments on the real value of testing:

> *The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration, and skills of those who design and produce the code. … But don't stop the tests: they are still essential to counteract the distracting effects and the perpetual pressure of close deadlines, even on the most meticulous programmers.*[36]

Another Turing Award winner, Edsger Dijkstra, states it this way:

> *Program testing can be a very effective way to show the presence of bugs but is hopelessly inadequate for showing their absence.* [37]

Because the states tested say nothing about the states not tested, no testing regimen short of exhaustive can give true confidence about the absence of undesirable behavior (see Section 4.2.1 for more information). Testing can demonstrate how software may benefit a mission but is not effective at gaining confidence that software cannot harm a mission (see Section 4.3.1 for further discussion).

Testing is effectively a sampling technique in which a small fraction of possible program inputs is selected to execute and monitor. A key reason for the popularity of fuzzing is its relative simplicity and effectiveness at finding certain classes of vulnerabilities, such as crashes. If a small sampling of the possible state space is effective at identifying undesirable behaviors (i.e., crashes), then this strongly suggests that the space unexplored by the testing may be similarly rife with undiscovered problems. The unexplored state space is typically many orders of magnitude larger than the state space that can be explored with dynamic testing. This means that the mission-threatening unexpected behavior that has not been discovered through testing is likely many orders of magnitude more than what has been discovered.

Many contemporary security products perform software testing by using special monitoring software to collect telemetry information on the test application or by running the application in an isolated sandbox test environment that has limited functionality. Any artificial testing environment introduces the risk that malicious modifications to software may be able to detect the artificial

---

[33] V. Manès, H. Han, C. Han, S. Cha, M. Egele, E. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," IEEE Transactions on Software Engineering (47), 2018: pp. 2312-2331.
[34] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," Communications of the ACM 52 (11), November 2009: pp. 74-84. https://doi.org/10.1145/1592761.1592781
[35] Input Space Coverage Matters, IEEE Computer, 53(1), January 2020, https://csrc.nist.gov/CSRC/media/Projects/automated-combinatorial-testing-for-software/documents/ieee-comp-jan-20.pdf.
[36] C. A. R. Hoare, "How Did Software Get So Reliable Without Proof?" https://6826.csail.mit.edu/2020/papers/noproof.pdf.
[37] Dijkstra, Edsger, "The Humble Programmer," Communications of the ACM 15 (10), October 1972: pp. 859–866.

environment and hide its malicious behavior. The software that fueled the Volkswagen emissions scandal did precisely this by detecting the standardized emissions tests (see Section 5.1). Malware has also become quite adept at detecting and hiding from sandbox environments.[38] In such cases, the tests will appear to be successful while malicious behavior is still present.

Testing is no substitute for software understanding. The recent paper "Systematic Review of Fuzzing" states:

> *One of the primary challenges [of fuzzing] is the limited comprehension of target programs, especially for complex programs. Fully comprehending the logic and data flow of programs is an arduous task. Consequently, this lack of comprehension impedes the exploration of in-depth paths within the program, thereby restricting the improvement of code coverage.*[39]

Rather than promising to narrow the software understanding gap, improved fuzzing strategies require an understanding of the program more deeply as a prerequisite to guide their explorations.

Although testing has value and should be continued, it is a weak alternative to software understanding and cannot sufficiently fill the software understanding gap. It can be useful in missions interested in finding *anything* of interest but is ill-suited to mission questions that depend more on finding *everything* of interest.

3.3.2 Digital Signatures

Digital signatures are cryptographically strong ways of ensuring that a file or program executable has not been changed since it was signed and for validating the entity that signed it. Digital signatures are often used to confirm that an executable came from a trusted vendor and serve as an alternative reason for trusting software even when its potential behaviors are not well understood.

Digital signatures unfortunately have several key shortcomings. Although they can provide strong guarantees that the content of the software did not change after it was signed, they make no statement about the behavior of the software itself. The SolarWinds incident is an example of this (for more information, see Section 5.2):

> *To recapitulate what is known so far: Intruders placed the so-called Sunspot malware into the SolarWinds system. The Sunspot malware was used to monitor and hijack the build process of the SolarWinds Orion app. This way, at compilation time, source code file content was replaced with a version containing the Sunburst malware.*
>
> *The resulting executable (called SolarWindsOrion.Core.BusinessLayer.dll) was digitally signed by SolarWinds…*[40]

---

[38] Clemens Kolbitsch, "Evasive Malware Tricks: How Malware Evades Detection by Sandboxes," *ISACA Journal*, vol 6, 2017, https://www.isaca.org/resources/isaca-journal/issues/2017/volume-6/evasive-malware-tricks-how-malware-evades-detection-by-sandboxes.

[39] Zhao, X., Qu, H., Xu, J. *et al.*, "A systematic review of fuzzing," *Soft Computing*, 2023, https://doi.org/10.1007/s00500-023-09306-2.

[40] Edilyn Teske, "The SolarWinds attack and best practices for code-signing," March 2021, Cryptomathic Blog. https://www.cryptomathic.com/news-events/blog/the-solarwinds-attack-and-best-practices-for-code-signing.

SolarWinds digitally signed the software, unaware that the software they were signing also contained malicious code. There are many other examples of signed software containing undesirable or malicious features.[41,42]

Furthermore, digital signatures do not ensure an executable came *from a trusted vendor*. The presence of the digital signature merely ensures that the executable was signed *by the signing certificate*—which could be borrowed, stolen, and even forged.[43]

For these reasons, although digital signatures have value and should be used, they are a weak alternative to software understanding and cannot fill the gap.

3.3.3 Malware Signatures

Malware signatures are used by anti-virus software to identify malware and block its execution before it can harm a computing system. Malware signatures are used as follows:

> *Although details may vary between packages, anti-virus software scans files or your computer's memory for certain patterns that may indicate the presence of malicious software (i.e., malware). Anti-virus software … looks for patterns based on the signatures or definitions of known malware.*[44]

Antivirus software is designed to recognize *known* malicious patterns. An initial analysis, often including manual review, is performed to understand whether the software is malicious and, if so, to identify the core details of its malicious behavior. Signature-based tools are then configured to detect the identified pattern in other software. Such approaches are fundamentally *retrospective* and cannot stop *future, novel attacks*.

The general-purpose nature of modern software (see Section 4.2.5 for more information) means there are an unbounded number of ways to make software malicious. Techniques that look for specific patterns perform poorly at recognizing novel malicious patterns. Most of the case studies discussed in Section 5 were not caught by antivirus scanners.

Consequently, although malware signature techniques have value and should be used, they are a weak alternative to software understanding and cannot fill the gap. Anti-virus tools would greatly benefit from the ability to more deeply understand the behavior of the software being analyzed.

3.3.4 Dynamic Software Monitoring

Dynamic monitoring software can be used to observe software as it runs on a system to detect certain undesirable or malicious behavior and instruct the operating system to block it before further harm occurs. Such approaches are more commonly found in enterprise and desktop environments. These techniques do not attempt to detect undesirable behavior before the software is placed into service, but rather seek to detect undesirable behavior as it occurs on a live system.

---

[41] Lawrence Abrams, "Microsoft-signed malicious Windows drivers used in ransomware attacks," Bleeping Computer, December 13, 2022, https://www.bleepingcomputer.com/news/microsoft/microsoft-signed-malicious-windows-drivers-used-in-ransomware-attacks/.

[42] Tom Spring, "Driver Disaster: Over 40 Signed Drivers Can't Pass Security Muster," Threat Post, August 10, 2019, https://threatpost.com/driver-disaster-over-40-signed-drivers-cant-pass-security-muster/147199/.

[43] See https://attack.mitre.org/techniques/T1649/ for a description of the Steal or Forge Authentication Certificates technique.

[44] CISA, "Understanding Anti-Virus Software," September 27, 2019, https://www.cisa.gov/news-events/news/understanding-anti-virus-software.

One key challenge with monitoring approaches in NS&CI mission spaces and systems is how to effectively respond when mission-threatening behavior is observed in the system. If the software is on a missile in flight, there are no good options. If the software operating the command, control, and communications network is found to have malicious behavior during a crisis, there are no good options. In many cases, mission success requires that software with mission-imperiling behavior not be deployed in the first place.

Detecting malicious behavior in general is a challenge, especially when the program under test is seeking to evade detection. This is not a currently solved problem.

For these reasons, although dynamic software monitoring has value and should be used, it is a weak alternative to software understanding because run-time detection is too late for many NS&CI missions. It does not sufficiently address the software understanding gap.

3.3.5 Reverse Engineering

Reverse engineering is the process of analyzing a finished technology product to determine how it was designed and how it operates.[45] Reverse engineers use information, insight, ingenuity, and extensive training to determine the parts of the executable to scrutinize or ignore, based on the specific mission question. In many circumstances, reverse engineers can achieve sufficient understanding of software to identify the technical evidence necessary to answer a given mission question for a given executable by leveraging a variety of tools to assist (e.g., disassemblers, decompilers, debuggers). Notably, reverse engineers can answer questions that cannot be answered using today's automated tools.

Of the other alternatives discussed in this section, reverse engineering and analysis is the best technique available today to understand the potential behavior of software.

Unfortunately, there are several disadvantages to human reverse engineering. First, a human's ability to engage in this process and understand software is limited by cognitive capacity. It is already the case that for many mission questions and much of modern software, the challenge is well beyond the human analyst's cognitive ability. Human cognitive ability is not scaling as fast as software development (see Section 2.3), resulting in an increasingly accelerating gap.

Second, the limited pool of adequately talented and trained individuals falls well short of what is needed to address the scale of the mission questions and software systems relevant to NS&CI today (see Section 2). Already insufficient, this pool cannot grow to meet future needs.

Third, human analysts are error-prone. As seen in the Therac-25 example above (Section 2.1.1), experts in a system can fail to find undesirable behavior in software, even when lives are at stake.

Lastly, human reverse engineering results vary widely. There is no standardized process or set of tooling that all analysts use; each analyst typically follows a different process and thereby identifies different evidence or may even arrive at different answers.

For these reasons, although human reverse engineering is one of the best software understanding alternatives available today and should continue, the process is too slow, expensive, error-prone, unreliable, and variable to be able to meet the ideal-state goal in which "mission owners are able to routinely ask any mission-related question of all mission-relevant software and receive rigorous,

---

[45] National Initiative for Cybersecurity Careers and Studies, "101 – Reverse Engineering," https://niccs.cisa.gov/education-training/catalog/federal-virtual-training-environment-fedvte/101-reverse-engineering.

reliable, rapid, and repeatable answers". Reverse engineering does not scale adequately to close the software understanding gap.

3.3.6 Mitigations

One approach used today is to introduce technical mitigations that reduce the potential for harm from certain classes of unexpected behaviors. Encrypting data at rest can mitigate the negative effects of data breaches. Software and hardware architectural mitigations increase the cost of software exploitation by reducing either the attack surface or the exploitation options for adversaries. Tightly controlled environments, Zero Trust architectures,[46] and numerous mitigations built into low-level processor architectures—such as Data Execution Prevention, Address Space Layout Randomization, Supervisor Mode Execution Prevention, Supervisor Mode Access Prevention, Structured Exception Handling Overwrite Protection, Control Flow Integrity, and Pointer Authentication Codes—are significantly increasing the difficulty of exploitation.

Such mitigations focus on a particular sub-set of mission questions related to violating relatively universal rules of math, computing or software programming (see Section 2.1.4). Violations of application-, mission-, or organization-specific rules are in different categories that current mitigation strategies are ill-suited to address. Further, many mission questions have nothing to do with vulnerabilities (see Appendix A). Although such mitigations have raised the difficulty and cost of software vulnerability exploitation, they have not made it impossible.

Although mitigations can slow and sometimes block certain negative effects from occurring, they do not assist mission owners in understanding the potential behavior of mission software. Responding to a divide-by-zero error by killing the offending program is valuable yet determining whether the program might be capable of attempting to divide by zero in the first place is an entirely different challenge. For many missions, knowing in advance what the program is capable of is key to mission success.

Although mitigations are very useful and have materially increased system security, when it comes to understanding the potential mission-threatening behavior of software, they are a weak alternative that can only address a small portion of the problem. Mitigations do not meaningfully address the software understanding gap.

3.3.7 Software Bill of Materials

A software bill of materials (SBOM) is a "formal record containing the details and supply chain relationships of various components used in building software."[47] An SBOM enables any software user to know the composition and dependencies of the software prior to using it. When vulnerabilities are discovered and publicized in a particular software package, SBOMs can enable a mission owner to rapidly determine if that same vulnerable software is incorporated into any mission systems.

Software producers often deliver software containing vulnerabilities (see the case study in Section 5.3). This occurs in part because producers often lack an understanding of the software contained in their product, which often leverages third-party libraries with dependency chains that include other

---

[46] National Institute of Standards and Technology (NIST), NIST Special Publication 800-207 "Zero Trust Architecture," https://doi.org/10.6028/NIST.SP.800-207.

[47] USG, Executive Order 14028, Section 10(j). https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-security-supply-chains-software-1.

software. If producers know the SBOMs of the software dependencies, they can avoid delivering software with known vulnerabilities that remain undetected in testing.

SBOMs could help mission owners know—prior to software acquisition—that the software is from a foreign source, which may be a material factor in key decisions. However, not all software from foreign sources is untrustworthy just as not all software from domestic sources is trustworthy. The true measure of software trustworthiness lies in its actual behavior, which SBOMs alone do little to address.

SBOMs are a useful tool. They are not, however, designed to provide sufficient understanding of potential software behavior to answer most mission questions (see Appendix A). Although they do reduce the software understanding gap in one way, most of the gap between the ideal state and current state in software understanding remains.

3.3.8 Policies and Procedures

Developers and companies adopt policies, practices, and technical regimes for software development to deliver a valuable product at an affordable market price. Market pressures alone have been insufficient to address the vulnerability aspect of the software understanding problem (see Section 2.1.3), and that is but one part.

In response, the USG is urging the private sector[48] to follow "Secure by Design" principles,[49] a set of recommendations for building technology products in a more secure manner.

Designing software to improve security from the start is a noble and worthwhile goal that has a chance to reduce the frequency of security-related unexpected behavior in software. However, while some mission questions involve traditional definitions of security, many do not. Secure by Design guidance is not intended to help mission owners understand software better. Furthermore, not every product on the market requires equal levels of security. NS&CI missions demand significant levels of rigor and support cost premiums that the household consumer market never will. Since the software development ecosystem extensively reuses software across a large variety of products, it is unclear to what extent Secure by Design activities will rise to the levels needed for the high confidence society expects of NS&CI missions.

Even if software providers rapidly and widely embrace Secure by Design activities and similar policies, the software understanding gap lies between the ability to create software and the ability to analyze it. Policies targeting software development practices are aimed at improving the software development side of the equation, not improving the software understanding side. Consequently, there would still be a deep software understanding gap even if better software development policies and procedures were widely adopted.

3.3.9 Attestation

Attestation is the process of a software vendor providing assurances to the government about a particular software product. Automotive manufacturers provide a *certificate of conformity* to attest that a given "engine or vehicle conforms to all application emissions requirements."[50] Software

[48] CISA et al., "Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software," Oct. 25, 2023, https://www.cisa.gov/resources-tools/resources/secure-by-design-and-default.
[49] "Secure by Design," Accessed Oct 29, 2023, https://www.cisa.gov/securebydesign.
[50] "Overview of Certification and Compliance for Vehicles and Engines," U.S. Environmental Protection Agency, https://www.epa.gov/ve-certification/overview-certification-and-compliance-vehicles-and-engines.

producers supplying the USG will soon be required to attest that they are following secure software development practices.[51]

Attestations are a solid basis for conformance and liability but not for understanding the potential behavior of software. Attestations are, however, insufficient to answer most mission questions (Appendix A). The case studies outlined in Section 5 involve a failure of the concept of attestation. Unethical or malicious vendors have a strong financial incentive to be evasive or lie while undetected malicious intrusions into a vendor's systems can render attestations inadvertently incorrect.

Although attestation has value and should be part of the overall software strategy, as a human trust-based mechanism, attestation provides a weak alternative that cannot help mission owners understand software well enough to build a technical evidence package regarding potential, mission-threatening behavior from that software. Attestation is not designed to close the software understanding gap.

3.4 Conclusions Regarding the Software Understanding Gap

To make properly informed decisions about risk mitigation or acceptance, mission owners need to know what inherent risks arise from potential behavior of a particular software product. Understanding this inherent risk is the first step in the risk assessment process. Failing to adequately characterize the inherent risk means that mitigations cannot be adequately applied, and the residual system risk cannot be properly assessed. A lack of adequate software understanding means that risk assessors cannot effectively implement the first step in the process, rendering the rest of the risk assessment process fundamentally flawed.

In an ideal state, to fully assess risk, mission owners would be able to routinely ask any mission-related question of all mission-relevant software and receive rigorous, reliable, rapid, and repeatable answers. These mission questions involve the potential for the software to place the mission at risk and go well beyond those routinely asked today. Compiling adequate answers to those questions requires understanding the software well enough to assemble a technical evidence package about its potential behavior relative to the mission question. As illustrated in the real-world example quoted in Section 2.3, some mission owners are posing important mission questions today that far outstrip our ability to answer.

In the current state, the importance of wrestling with the challenges posed by software is widely understood, as illustrated by the broad set of alternatives being pursued today. Each of the alternatives listed above was adopted to address some kind of gap in the use of software in important missions. Although testing, digital signatures, malware signatures, dynamic software monitoring, reverse engineering, mitigations, SBOMs, policies and procedures, and attestation approaches all have their respective value, only reverse engineering is suitable to helping mission owners understand software adequately. Unfortunately, with presently available tools, reverse engineering remains a heavily manual activity that cannot scale to the scope of the accelerating problem.

Consequently, the capabilities available today for creating software vastly outstrip the capabilities available for understanding it well enough to answer vital NS&CI mission questions. *This is the software understanding gap.* In many cases, sustained lack of capability to bridge this gap has already produced a state of learned helplessness in which government leaders and NS&CI mission owners

[51] U.S. Office of Management and Budget, "Memorandum for the Heads of Executive Departments and Agencies," M-22-18, September 14, 2022, https://www.whitehouse.gov/wp-content/uploads/2022/09/M-22-18.pdf.

have learned to down-scope their expectations and mission questions to match the capabilities at hand. Even in NS&CI mission spaces, software is placed into service which cannot be adequately analyzed. Mission owners opt to deploy this software because they assess that the benefits of doing so outweigh the risks, even though the risks are unmeasurable and unexpected software behavior has not only led to mission failure in the past but also has the potential to lead to national-scale catastrophe. They have no better alternative.

A rich, extensive, growing ecosystem of software development tools and libraries with increasing levels of automation is fueling the accelerating software production curve in Figure 2. As a result, software is becoming more prevalent throughout government and society, is becoming larger and more complex, and is being produced faster. The current understanding gap is not just growing, it is growing at an accelerating rate. This is not a future challenge; it is the reality today. Closing this gap requires an ecosystem of analysis components, libraries, frameworks, and tools that mirror the ecosystem driving software production, designed for rigorous, reliable, rapid, and repeatable software understanding. This type of software understanding ecosystem does not exist today and no activity is on track to create it.

The next section traces some key historical elements to explain the factors that have led to this present situation and examines the characteristics of software that make it challenging to understand.

4 The History and Challenge of Understanding Software

It may be natural to wonder why the software understanding gap discussed in the previous sections exists in the first place. Why is it hard to understand software? How did we come to depend so broadly on something which now threatens our most important national responsibilities?

The risk to NS&CI missions posed by the software understanding gap did not arise overnight. Like the proverbial frog in the pot, decisions made in the past, which were reasonable at the time, have rendered our current situation perilous due to evolving trends. This section will explore two primary reasons for the present situation.

First, several trends have changed humanity's use and understanding of software over the past 80 years, including the plummeting cost and rising capacity of computing hardware, proliferation of software, accelerating interconnectivity, and growing understanding of software exploitation. Choices that were reasonable based on what was known in past decades have not withstood the test of time. Section 4.1 will trace this history.

Second, the software understanding gap is rooted in the unique combination of the characteristics of software systems that makes understanding potential behavior technically challenging. Software systems are unlike any natural phenomenon in daily experience, causing our intuitions as humans to lead us to incorrect conclusions. Section 4.2 will describe these characteristics in more detail.

Section 4.3 ends with several implications and conclusions, drawn from the history and characteristics of software, which provide insight into the software understand gap. That section explains why testing is not a solution to the software understanding gap, why predicting potential behavior of software is so challenging, and why tremendous progress in software understanding can be possible despite strong theoretical findings that fully automating perfect program understanding is impossible.

4.1 Historical Choices that Drive the Software Understanding Gap

There have been decades of policy choices that led to the present situation and its challenges. This section briefly traces the history of computing and software to explain, first, how those policy decisions made sense at the time, and second, how the changing nature of software has compromised the assumptions underpinning those decisions and led to the software understanding gap and the resulting NS&CI mission risk.

4.1.1 The Invention of Computing and Software

The birth of the modern computing era is widely attributed to Alan Turing, who, during the 1930's while working on his Ph.D. at Princeton University, published a paper describing a theoretical automatic "universal computing machine," which operates over 0's and 1's.[52] While many algorithms exist for computing a variety of things, the Universal Turing Machine (UTM) can be used to compute anything which can be computed, merely by changing the contents of a tape that directs the actions of the machine—later generations would call the contents of the tape the "program." The UTM was a mathematical notion, perhaps inspired by automated machines of the previous century, but not physically constructed until much later.

Inspired by this theoretical work, the first physical implementations of the computer would appear in the 1940's, created by physicists and mathematicians to replace "human computers" with machines to accelerate long sequences of calculations.[53] Early computers were hard-wired to perform a single algorithm, as the following quote demonstrates:

> *ENIAC's main drawback was that programming it was a nightmare. In that sense it was not a general use computer. To change its program meant essentially rewiring it, with punchcards and switches in wiring plugboards. It could take a team two days to reprogram the machine.*[54]

Although they required a lot of effort to construct, early computers had very limited complexity. The ENIAC used roughly 17,000 vacuum tubes and 1,500 relays for its main operations. It weighed more than 27 tons, took up 1800 square feet, and consumed 150 kW of power.[55] All of that was necessary to perform just a handful of mathematical operations on just twenty 10-digit decimal numbers. The main advantages of the ENIAC came from its speed, performing 5000 additions per second, and its ability to do repetitive calculations 24 hours a day, 365 days a year. Due to the difficulty of reconfiguring the ENIAC to run a new program (it had approximately 6000 manual switches),[56] much thought was given to simplifying the problem so that it would fit within the very limited computing capabilities of the machine. The designers and programmers were able to analyze and understand that the programs they were creating were correct.

In the 1950s, improvements were made to remove the need to rewire the machine for new algorithms. The terms "software" and "hardware" were introduced to distinguish computer programs from computers. Thus, the first universal computing systems came on the scene that no longer required rewiring for a new algorithm. By changing the data alone, without any

---

[52] Turning, Alan. (1936) "On Computable Numbers, With an Application to The Entscheidungsproblem," https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf.
[53] Steven Levy, "The Brief History of the ENIAC Computer," *Smithsonian Magazine,* November 2013.
[54] PBS, "ENIAC is built" (1945), https://www.pbs.org/wgbh/aso/databank/entries/dt45en.html.
[55] University of Michigan, Computer Science and Engineering, "ENIAC Display" https://cse.engin.umich.edu/about/beyster-building/eniac-display/.
[56] Brian J. Shelburne, "The ENIAC at 70," *Math Horizons* 24, no. 3 (2017): 26–29, https://doi.org/10.4169/mathhorizons.24.3.26.

reconfiguration of the hardware, one can change what the computer does. That is, one machine can implement almost any algorithm of computation, from missile trajectories to tic-tac-toe.

That same decade saw the invention of the first high-level programming languages, which freed the programmer from the need to write the computer program in terms the computer best understood—1's and 0's—and empowered them to develop programs using familiar formulas. Over time, programming languages would enable human developers to express ever-increasingly complex algorithms in simple and natural ways.

The invention of the transistor at Bell Labs in 1947 and the first transistorized computer at the Massachusetts Institute of Technology (MIT) in 1956 would radically reduce the size and power requirements of the computer, while increasing its speed and enabling it to store much larger programs than ever before. In 1965, Gordon Moore, co-founder of Intel Corporation, noticed that the number of transistors on a single integrated circuit was doubling every 18 to 24 months—an observation that came to be called *Moore's Law.* Such exponential growth was inconceivable in earlier decades when the vacuum tube and relay were the building blocks. These inventions led to decades-long exponential increases in computing power that has never been seen in any other field in history. The exponential growth in computational power fueled a similar exponential growth in the complexity of the programs that such computers could execute.

Reflections on Software Understanding. In the beginning days of computing, computers were massive and excessively expensive. Each machine, being unique, had custom programs developed for it specifically, hand-crafted to its capabilities and idiosyncrasies. Physical security from guards, gates, and guns was a fraction of the price of the machine itself, the space it occupied, and the power it consumed. The term *computer bug* was first coined in 1947 when a dead moth was found to be blocking a relay—unexpected behavior in the programming most frequently arose from failures in the hardware. Computers were isolated from each other with no ability to communicate (it was not until 1959 that the first commercial modem would be released by Bell Telephone). Program complexity was strictly limited by the physical limitations of the machine, arising out of its electro-mechanical design around vacuum tubes, relays, and toggle switches. The algorithms being executed on the computers were readily understood by experts.

4.1.2 Expanding Use of Software

Fueled in large part by the invention of the transistor, during the coming decades, computing hardware shrank in size and cost exponentially, while growing in capacity similarly. Overall capability per cost improved by more than a million-fold, enabling ever larger programs.

1970s. As early as the 1970s, concerns were being raised that the complexity of software was rapidly outstripping the ability of programmers to understand it. In 1972, in his Turing Award acceptance article, Edsger Dijkstra said of software programming:

> *The major cause [of the software crisis] is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic*

> *industry has not solved a single problem, it has only created them, it has created the problem of using its products.* [57]

That same year, the Electronic Systems Division of the Air Force released the "Computer Security Technology Planning Study"[58]. The report was focused on "how to provide multilevel[59] resource and information sharing systems secure against the threat from a malicious user". It noted:

> *There is little question that contemporary commercially available systems do not provide an adequate defense against malicious threat. Most of these systems are known to have serious design and implementation flaws that can be exploited by individuals with programming access to the system. … The design and implementation flaws in most contemporary systems permit a penetrating programmer to seize unauthorized control the system, and thus have access to any of the information on the system.*

Limited connectivity among computer systems meant that "individuals with programmatic access to the system" was a relatively small group of people. The report went on to describe several types of software vulnerabilities, noting that "there is no way to certify (guarantee) that one or more 'trap doors' have not been inserted in the operating system" by the manufacturer.[60]

1980s. A principal challenge in the 1980s became learning how to harness this burgeoning computing power by improving software engineering approaches and practices. Renowned software engineering expert and Turing Award winner Fred Brooks published an article in which he described the problem:

> *The familiar software project…is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.*[61]

Due to the increasing challenge in rapidly creating reliable software, Fred Brooks offered the following as his first "promising" avenue for improving the software engineering situation:

> *Buy versus build. The most radical possible solution for constructing software is not to construct it at all. Every day this becomes easier, as more and more vendors offer more and better software products for a dizzying variety of applications. … Any such product is cheaper to buy than to build fresh.*

---

[57] Edsger Dijkstra, "The Humble Programmer", Communications of the ACM 15 (10), October 1972: pp. 859–866.
[58] Anderson, James P., "Computer Security Technology Planning Study," Air Force Electronic Systems Division, ESD-TR-73-51, October 1972.
[59] A "multilevel" system is a single computer that can support users and information at different levels of classification with strong guarantees that users cannot access information to which they are not authorized. Given the relatively high cost and limited numbers of computers in the 1970's, both economic and operational pressures led to the exploration of such systems.
[60] A *trap door* is an unexpected behavior added to software. A related term is *backdoor*, which is often used to refer to bypassing authentication checks. Both are secretive modifications to software. Often, they are inserted with malicious intent, but there are cases where vendors have added such modifications to software to facilitate maintenance or customer service.
[61] Brooks, Frederick P., "No Silver Bullet—Essence and Accidents of Software Engineering," Paper presented at the meeting of the Information processing 86, North-Holland, 1986.

This recommendation would shortly find its way into government policy, as described below. Regarding complexity, he remarked:

> *Software entities are more complex for their size than perhaps any other human construct.*

In this same timeframe, in his Turing Award acceptance article, Ken Thompson made a revelation with profound implications for computer security in coming decades. Thompson described the clever way in which he had inserted a secretive modification into the software he was developing that would bypass authentication when logging in to the UNIX computer system. Ingeniously, the modification was not placed in the source code for the `login` program but was added to the executable by the compiler. The code to instruct the compiler to do this was also not present in the source code of the compiler but was self-propagating from one compiler executable to the next. The entire sequence was hidden from any audit of the source code.

The innovation of high-level programming languages in the 1950s had created a degree of separation between the programmer's intentions and what was actually running on the computer—it was in this separation that Thompson's subversive changes hid. From this, Thompson concluded:

> *The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code.*[62]

Unfortunately, awareness of such nefarious behavior potentially lurking in executables from third-party developers would remain limited to largely academic circles for many years.

In the 1980s, plummeting computer costs and rising capabilities enabled both business and home users to own computers to accelerate daily tasks. Large businesses with custom needs would still retain programming staff to create custom software. But home users and small business owners, lacking programming expertise, would purchase software for their various needs, fueling new markets for third-party software.

1990s. In 1994, Secretary of Defense William J. Perry issued a memo directing the Department of Defense to pivot away from military specifications to commercial standards.[63] Turning from Government Off-The-Shelf (GOTS) to Commercial Off-The-Shelf (COTS) was expected to provide many benefits, not the least of which would be reduced cost and increase reliability. This was not expected to result in an increase to mission risk, but rather to reduce it.

In 1996, one of the seminal articles on how to take advantage of a software flaw for malicious purposes appeared in a hacker e-magazine named *Phrack*. Entitled "Smashing the Stack for Fun and Profit,"[64] it inspired many to explore the art of software exploitation. In 1998, *Phrack* documented the first database exploit.[65]

In 1999, the MITRE Corporation launched the Common Vulnerability Enumeration (CVE) project,[66] which records and archives third-party reports of software vulnerabilities. In 1999, the CVE archive recorded 321 reported vulnerabilities. In 2000, the number quadrupled to 1,438.

---

[62] Ken Thompson, "Reflections on Trusting Trust." Communications of the ACM 27(8): 761-763 (1984).
[63] William J. Perry, "A New Way of Doing Business". 29 June 1994. https://www.sae.org/standardsdev/military/milperry.htm
[64] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, volume 7, issue 49, 31 Dec 1995.
[65] http://phrack.org/
[66] MITRE Common Vulnerabilities and Exposures, Accessed Oct 29, 2023, https://www.cve.org/.

During this era, interconnectivity and communication began to accelerate. The first commercially available modem for connecting computers via phone lines was brought to market by Bell Telephone in 1959; plummeting costs would make them commonplace over the next two decades. The ARPAnet project, begun in 1969 by DoD's Advanced Research Projects Agency (DARPA), leveraged dedicated connections links to connect research institutions across the United States. In 1981, the National Science Foundation (NSF) began leveraging ARPAnet concepts to create the Computer Science Network (CSNET). This connected computer scientists at numerous universities and research institutions. CSNET would be phased out in 1995 with the rise of the commercial internet. By the year 2000, 84% of American adults reported having used the internet and 1% of households had full-time broadband access.[67,68]

Reflections on Software Understanding. The rapidly accelerating hardware capabilities and plummeting costs would begin the process of society's digital transformation. Widespread adoption of computers created whole new markets for business and home software, created not by the expert end users as in the beginning days of computing, but by professional software development companies. Although the Therac-25 (see Section 2.1.1), Ken Thompson, and others demonstrated the potential of unexpected behavior, these examples were not widely known, and the full effects of unexpected behavior were yet to be realized. The widespread interconnectivity meant that guards, gates, and guns were no longer sufficient to protect the computer from malefactors, but problems arising from this were very limited. During this timeframe, software complexity grew so rapidly that many struggled to make reliable software systems at reasonable costs, leading to Fred Brooks' recommendation to buy vs. build. These trends resulted in the enormously influential decision to pivot DoD acquisition to commercially available options when possible.

4.1.3 The Modern Era: Software Is "Eating the World"

During this era, despite growing awareness of software vulnerabilities and other flaws, the creation and adoption of software into more facets of society would continue to accelerate. In 2011, Marc Andreessen, co-founder of Netscape (which produced the first widely popular web browser), wrote:

> *Software is Eating the world… Six decades into the computer revolution, four decades since the invention of the microprocessor, and two decades into the rise of the modern Internet, all of the technology required to transform industries through software finally works and can be widely delivered at global scale.* [69]

Widespread interconnectivity of the world's computers through the internet continues to accelerate. In 2018, the "Internet of Things" (IoT) reached 7 billion devices connected to the internet. In 2019, fifth generation (5G) mobile networks were launched, promising to bring revolutionary connectivity and smart cities. In 2021, three quarters of American households had broadband connectivity and over 4.6 billion people globally were connected to the internet, representing over half the global population.

The rise in ubiquitous interconnectivity has provided immeasurable economic benefit, enabling instantaneous communication around the globe. Along with a period of relative global stability, this

---

[67] Kim Ann Zimmermann, Jesse Emspak, "Internet history timeline: ARPANET to the World Wide Web," Live Science, April 8, 2022, https://www.livescience.com/20727-internet-history.html.

[68] Andrew Perrin; Maeve Duggan, *Americans' Internet Access 2000-2015* (Pew Research Center, 2015) https://www.pewresearch.org/internet/2015/06/26/americans-internet-access-2000-2015/.

[69] Marc Andreesen, "Why Software Is Eating the World," AH Capital Management, L.L.C. ("a16z"). (originally published in The Wall Street Journal, August 20, 2011), https://a16z.com/why-software-is-eating-the-world.

connectivity has fueled globalization in many markets, including software development. Ease of sharing code around the world has fueled an expanding ecosystem of advanced software development tools. Software manufacturers are increasingly developing software by leveraging a vast array of pre-existing components and libraries, fused together to make the final software product. Maturing software development tools and an extensive network of volunteer-driven communication forums are enabling far less experienced programmers to contribute to the ecosystem.

Ubiquitous, global connectivity brings with it a new threat. Whereas once a malefactor seeking illicit access to a computer would have required physical access, now such individuals can access billions of computing devices from anywhere on the globe. Coupled with widespread knowledge of how to exploit unintentional software flaws, cyber-attacks from individual users, organized crime, and nation-states have begun to rise. Such attacks occur not only via direct connection but also by inserting malicious code in the software supply chain.

In 2017, on the eve of the Ukrainian Constitution Day, the Russian military[70] mounted a software supply chain attack by injecting malicious code into the public release of a widely popular commercial Ukrainian tax software program, "M.E.Doc." Dubbed "NotPetya," the attack and its effects were reported in WIRED magazine:

> *Within hours of [NotPetya's] first appearance, the worm raced beyond Ukraine and out to countless machines around the world, from hospitals in Pennsylvania to a chocolate factory in Tasmania. It crippled multinational companies including Maersk, pharmaceutical giant Merck, FedEx's European subsidiary TNT Express, French construction company Saint-Gobain, food producer Mondelēz, and manufacturer Reckitt Benckiser. In each case, it inflicted nine-figure costs. It even spread back to Russia, striking the state oil company Rosneft.*
>
> *The result was more than $10 billion in total damages…*[71]

It is worth noting that, in order to spread and cause the amount of damage it did, the NotPetya supply-chain attack software evaded malware scanners and other network security mitigations around the world. None of the companies impacted had the capabilities necessary to analyze the update and detect its malicious nature prior to putting it into use.

In 2020, a software supply chain attack—attributed to the Russian Foreign Intelligence service—would modify the SolarWinds' Orion software platform (see Section 5.2 for more detail) to target numerous USG agencies and companies.

While it is difficult to quantify, the global annual cost of cybercrime is estimated to be $6-$8 trillion per year.[72,73] Much of this crime is possible because victims lack the capabilities necessary to detect the ways in which software may behave unexpectedly or maliciously. Poor understanding of

---

[70] CISA, "Petya Ransomware," Feb 15, 2018, https://www.cisa.gov/news-events/alerts/2017/07/01/petya-ransomware.
[71] Andy Greenburg, "The Untold Story of NotPetya, the Most Devastating Cyberattack in History," *WIRED Magazine*, Aug 22, 2018, https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.
[72]Purple Sec "Cost of Cybersecurity" https://purplesec.us/resources/cyber-security-statistics/#Cost.
[73] Juniper Research, "Cybercrime to Cost Global Business over $8 Trillion in the Next 5 Years," 2017, https://www.juniperresearch.com/press/cybercrime-to-cost-global-business-over-8-trn.

software affects more than crime—estimated costs to the U.S. in 2022 alone from poor software quality was $2.4 trillion.[74]

In 2021, the White House released EO 14028, "On Improving the Nation's Cybersecurity,"[75] in response to what it described as the "persistent and increasingly sophisticated malicious cyber campaigns that threaten the public sector, the private sector, and ultimately the American people's security and privacy."

Reflections on Software Understanding. In stark contrast to the first era of computing, the modern era ushered in a shift to the use of third-party software, both commercial and open source. The vast majority of the population of the globe uses software-driven computing, communication, banking, government services, IoT devices, and more. Computing has revolutionized NS&CI missions, transforming their systems to be software-driven. An infinitesimal fraction of all this software is built by or deeply understood by those who will use it. Awareness of the vulnerabilities and unexpected software behavior foreshadowed by earlier eras has come of age, touching nearly every segment of society and government through malware, ransomware, supply chain attacks, or mere flaws that make software unreliable. Hardware capacity continues its meteoric rise while prices continue to fall, fueling a similar meteoric rise in the complexity of software. The European Space Agency's Automated Transfer Vehicle is driven by an estimated 1 million lines of code, automobiles now have hundreds of millions of lines of code, and the software that powers Google's various services is estimated at over 2 billion lines of code.[76,77]

In all this complexity, the total number of vulnerabilities reported by MITRE's CVE system has grown over the years, with over 25,000 reported in 2022 and a program lifetime total of nearly 200,000. That number only represents known vulnerabilities—there is no way of knowing the fraction of the total number of vulnerabilities this represents.

The tremendous economic benefits from adopting commercial software, anticipated by both Fred Brooks and William Perry, conflict with the tremendous risks and costs from unexpected behavior in software that is used in critical missions despite being inadequately understood.

4.2 Software Characteristics that Drive the Software Understanding Gap

The history related above highlights the key points that helped shape the evolution of software development. This section describes the characteristics of software, many arising out of its history, which make it so challenging to understand.

Modern software systems are radically different than everything else in daily human experience. There is a useful parallel in a quote that Richard Feynman used regarding electrons:

> *[Electrons] behave in a way that is like nothing that you have seen before. Your experience with things that you have seen before is incomplete. The behavior of things on a very tiny scale is simply different. An atom does not behave like a weight hanging*

[74] Consortium for Information & Software Quality, "The Cost of Poor Software Quality in the US: A 2022 Report," 2021, https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/.
[75] USG, Executive Order 14028, "On Improving the Nation's Cybersecurity," https://www.govinfo.gov/content/pkg/DCPD-202100401/pdf/DCPD-202100401.pdf.
[76] "Million lines of code making ATV Europe's smartest spacecraft (ESA)," European Space Agency, https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Million_lines_of_code_making_ATV_Europe_s_smartest_spacecraft.
[77] David McCandless, Pearl Doughty-White, Miriam Quick, "Codebases – Millions of lines of code," Sept 24, 2015, https://informationisbeautiful.net/visualizations/million-lines-of-code/.

> *on a spring and oscillating. Nor does it behave like a miniature representation of the solar system with little planets going around in orbits. Nor does it appear to be somewhat like a cloud or fog of some sort surrounding the nucleus. It behaves like nothing you have seen before.*[78]

Modern software systems, like electrons, are like nothing you have seen before. Counter-intuitively, software *seems* easy to understand, since much of it works as expected most of the time, but our ability to create software that functions well much of the time vastly outstrips our ability to understand the ways in which it may fail to function and what the ramifications of those failures may be. *This is the software understanding gap.*

Modern software systems, even relatively simple ones, are run on modern computing hardware with various embedded firmware images, managed by an operating system. Further, modern software systems are built with compilers, programming languages, libraries, and much more. Although a given software application may be quite simple, virtually without exception, the computing system around it is far from simple. For an NS&CI mission owner, it is the performance of the overall computing *system* that is necessary for mission success.

NS&CI missions and systems require the highest levels of confidence. As NS&CI systems have become transformed from the physical systems of decades past to the software-driven systems of today, it is imperative that the risk to NS&CI missions arising from software be understood. Unfortunately, predicting when software may fail and in what way, determining whether it does or does not have any malicious behavior inserted into it, and knowing the ways in which it may threaten the missions which depend on it, is as difficult as it is important. This section explains that difficulty.

The following sub-sections each briefly describe one of the following characteristics of software that make it challenging to reason about:

- Discrete
- Dynamical
- Non-linear
- Complex
- Universal

There is a final characteristic that does not make software challenging to understand, but which makes it so useful for a wide range of applications while also enabling software attackers to exploit software systems more reliably. This characteristic is:

- Deterministic

The above characteristics collectively explain why software has unexpected behavior, why it is challenging to analyze, and what types of capabilities are needed to reduce the software understanding gap.

4.2.1 Discrete vs. Continuous

To oversimply slightly for explanatory reasons, *continuous* phenomena are those that change smoothly, with continuous variation and no sudden, instantaneous jumps in state. The temperature in a room, the volume of water in a glass, and the speed of an object are all examples of things that change smoothly and continuously over time. The speed of a car cannot jump instantaneously from

[78] Richard Feynman, *The Character of Physical Law*, 1967.

0 MPH to 100 MPH. Most of our experiences in the physical world involve continuous phenomena, most of which can be modeled by smooth curves in mathematics. Graphs of continuous phenomena do not have breaks or jumps.

In a continuous system, interpolation can be used to reasonably predict behavior between two known points. For example, if a bridge can support a small weight, A, and a heavy weight, B, it can be safely assumed that it will support all weights between A and B, all other things being equal. Note that there are an infinite number of weights between A and B. In a continuous system, when there is confidence that all points between A and B lie within the nominal behavior of the system, it is not necessary to test each of these infinite possibilities. Stated differently, within nominal operating ranges, continuous systems do not have sudden, instantaneous, wildly unexpected points. Consider filling a glass with water—when a tiny bit is added, the cup gets a tiny bit fuller. It would be entirely unexpected that at a certain, exact point of fullness (say 62.8723957%), adding a tiny bit of water would result in the flooding of the entire room.

In contrast, *discrete* systems are those that have sudden jumps between states. The temperature of a room may be continuous, but the number of people in it is discrete. There is not an infinite number of people between 1 and 2 as there is with weights, temperature, etc. Graphs of discrete systems do have breaks or jumps.

Discrete systems cannot safely bear interpolation between two points—intermediate points may exhibit wildly different behavior. Testing a system with inputs "1" and "3" tells you nothing about the behavior on input "2." As an example, if a computer login screen rejects both "Aardvark" and "Zythophile," users understand enough about such systems to not expect that every possible word in between will also be similarly rejected. Users expect that there is an input, the proper password, which will result in radically different behavior than every other input.

Modern software systems are discrete, not continuous. Furthermore, software can make active decisions about how and when to be discontinuous. Consequently, testing software at points A and B tells you nothing for certain about points in between. This fact is echoed in a report by the National Research Council regarding software in dependable systems:

> *Since software lacks the continuity of physical systems that allow inferences to be drawn from one sample execution about neighboring points, testing says little or nothing about the cases that were not exercised.*[79]

4.2.2 Dynamical vs. Static

The difference between *dynamical* (also called "dynamic") and *static* systems is explained by Smarajit Ghosh in "Signals and Systems" as follows:

> *The system is said to be static if its output depends only on the present input. On the other hand, if the output of the system depends on the past input, the system is said to be dynamic.*[80]

A static system has no memory of the past. A fair coin toss has no memory of how many times it came up heads in the past. Rain falls to the ground based on current conditions, without regard to past conditions.

---

[79] National Research Council, *Software for Dependable Systems: Sufficient Evidence?* (National Academies Press, 2007).
[80] Smarajit Ghosh, *Signals and Systems*, (Person India, 2005).

In contrast, a dynamical system "remembers"—to predict a dynamical system's future behavior, the past states must be known. The population in a herd of deer each year is influenced by the population the previous year. The metal in a paperclip changes as it is bent, which it can withstand only so many times before it breaks. The memory of a dynamical system makes it fundamentally more challenging to analyze, because, in addition to its present inputs, some knowledge of the system's past is required to predict its behavior.

Modern software systems are dynamical. The user expects the website to remember the new password, word processor to remember the text typed into it, and the bank software to remember the deposit. There are a few limited circumstances that call for software with no memory, producing the same output given the same input each time, such as a software library for computing mathematical functions. Even if a software application is carefully written to be static, the computing platform and operating system, upon which it runs, are dynamical, which can affect its operation.

4.2.3 Non-linear vs. Linear

In a *linear* system, output is proportional to input. Taxi fares are typically linear—after paying an initial price, there is a fixed cost for every mile driven. An unwaveringly steady drip will fill a glass at a linear rate. The distance covered by a vehicle will grow at a linear rate if the vehicle's speed remains fixed. Graphs of linear phenomena make straight lines.

Non-linear systems do not make straight lines when graphed. The trajectory of a missile, compounding interest, and anything that accelerates or decelerates are examples of non-linear behavior that are relatively simple to characterize. The weather, the movement of ocean currents, and explosions are also examples of non-linear behavior, but ones that are much more complex. As Christian Willy and others explain it:

> *In [non-linear] systems there exists no proportionality and no simple causality between the magnitude of responses and the strength of their stimuli: small changes can have striking and unanticipated effects, whereas great stimuli will not always lead to drastic changes in a system's behavior.*[81]

The exponential growth of computing and software are non-linear phenomena. The original idea of the Universal Turing Machine was to design a machine capable of computing anything that could be computed, including non-linear algorithms. Some of the earliest programs were in fact designed to perform calculations of missile trajectories and modern software is used to predict the weather, ocean currents, explosions, and much more.

Modern software systems are non-linear.

4.2.4 Complex vs. Simple

Unlike *simple* systems, whose behavior can be readily understood, *complex* systems behave in ways that cannot be understood simply by examining their constituent parts. For example, understanding hydrogen (H) and oxygen (O) in isolation does not lead to an understanding of water (H_2O)—the "system" behaves in ways that neither of the components ever do. The term often used for this is *emergent* behavior.

---

[81] Christian Willy, Heinz Gerngroß, Edmund A.M. Neugebauer, The Concept of Nonlinearity in Complex Systems. *European Journal of Trauma and Emergency Surgery*, 11–22, 2003, https://doi.org/10.1007/s00068-003-1248-x.

The difference between simple and complex systems is described in “Complexity Explained” as follows:

> *In simple systems, the properties of the whole can be understood or predicted from the addition or aggregation of its components. In other words, macroscopic properties of a simple system can be deduced from the microscopic properties of its parts. In complex systems, however, the properties of the whole often cannot be understood or predicted from the knowledge of its components because of a phenomenon known as “emergence.” This phenomenon involves diverse mechanisms causing the interaction between components of a system to generate novel information and exhibit non-trivial collective structures and behaviors at larger scales. This fact is usually summarized with the popular phrase the whole is more than the sum of its parts.*[82]

This complexity arises from the interaction of the various parts or components in a complex system, as explained in “Dynamics of Complex Systems”:

> *A dictionary definition of the word “complex” is: “consisting of interconnected or interwoven parts.” Why is the nature of a complex system inherently related to its parts? Simple systems are also formed out of parts. To explain the difference between simple and complex systems, the terms “interconnected” or “interwoven” are somehow essential. Qualitatively, to understand the behavior of a complex system we must understand not only the behavior of the parts but how they act together to form the behavior of the whole. It is because we cannot describe the whole without describing each part, and because each part must be described in relation to other parts, that complex systems are difficult to understand. This is relevant to another definition of “complex”: “not easy to understand or analyze.”*[83]

As explained by Fred Brooks, the nature of how software is constructed makes complexity an essential property:

> *Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike... If they are, we make the two similar parts into one, a subroutine, open or closed. In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound... The complexity of software is an essential property, not an accidental one.*[84]

Modern software systems are complex.

4.2.5 Universal vs. Particular

In the age of the blacksmith, each object the blacksmith created was designed to do a particular set of tasks. Pounding, cutting, digging, prying, securing, and so on are such different activities that each call for its own specialized tools. In the physical world, a single universal tool that could do everything that a person could imagine has never been invented, nor is it likely to be invented.

---

[82] M. De Domenico, D. Brockmann, C. Camargo, C. Gershenson, D. Goldsmith, S. Jeschonnek, L. Kay, S. Nichele, J.R. Nicolás, T. Schmickl, M. Stella, J. Brandoff, A.J. Martínez Salinas, H. Sayama. Complexity Explained (2019). DOI 10.17605/OSF.IO/TQGNW
[83] Y. Bar-Yam, *Dynamics of Complex Systems*, (Westview Press, Boulder, 1997).
[84] Frederick Brooks, "No Silver Bullet—Essence and Accidents of Software Engineering." Paper presented at the meeting of the Information processing, 1986, North-Holland.

In contrast, when it comes to processing information, modern microprocessors are a form of *universal* machine—a machine able to execute any algorithm that can be possibly expressed in software, subject only to practical limits of the computing device. By merely changing the software that it runs, the same modern computing device can act as a missile guidance system, a video game controller, a database manager, a secure communications link, and many more. Universality is a powerful concept that has paid off handsomely, enabling electronics companies to mass-produce extremely versatile microprocessors at extremely low cost.

Universality also comes with drawbacks. It provides all the building blocks that a software creator could ever need to design any program but may inadvertently enable an adversary to access those same building blocks. Vulnerabilities and mistakes in software executing on universal machines have a much greater potential for devastating impact than if the underlying hardware architecture was limited to a particular function. Because universal machines can do anything, unintended and undesirable behaviors in those machines can do anything. Consequently, exploiting vulnerabilities in modern software often results in complete system compromise because the attacker can use all the functionality of the universal machine to accomplish their goals, rather than being constrained to the limited functionality of a particular system.

Because of the economies of scale, market pressures drive developers toward universal solutions that greatly exacerbate the software understanding gap. Universal computing hardware and software are typically many orders of magnitude less expensive than those specially designed for a particular use. This is true for computer processors and other hardware, operating systems, computer languages, software components and libraries, and much more. These economies of scale drive plummeting prices, which, in turn, drive mission software systems to leverage cheap, universal computing components, which are then adapted for particular use through software. The underlying universal nature of the components that comprise such systems introduces complexities that make the resulting software dramatically more challenging to understand, greatly expanding the software understanding gap.

Modern software systems are run on universal computing hardware, built with universal languages by universal compilers to run on universal operating systems.

4.2.6 Deterministic vs. Stochastic

In a deterministic system, each time the system is given the same input in the same state, it will produce the same output. When you dial 911 from a cell phone, you expect it to do precisely that every time, and not to instead dial a random number. Each time you place a charge on your credit card, you expect exactly the charged amount to be reported on your bill and for your bill not to contain charges from different accounts. Algebra is a deterministic system—2 plus 2 will always equal 4.

In contrast, stochastic systems may produce varying outputs, even when given the same input in the same state. A fair coin flip produces a stochastic result. Whether a batter will hit a homerun is a stochastic event. The math of probabilities is used to describe stochastic phenomena, which may follow a variety of distributions, such as a bell curve or a flat, uniform distribution.

With rare exception, computer hardware and software are carefully designed to be deterministic systems. Just as in algebra, when the computer adds 2 and 2, great care is taken to ensure the answer is always 4. When you enter a password into a website, you do not expect it to make a probabilistic decision as to whether to authenticate you, but instead to authenticate you if and only if the password entered is correct. In nearly every case, stochastic behavior in computing systems either deterministically responds to stochastic effects in the real world (such as the arrival of input

into the system), or merely has the appearance of being stochastic while being based on deterministic phenomena (such as a pseudo-random number generator).

This characteristic of determinism does *not* contribute to the software understanding gap; in fact, it helps ameliorate it slightly in the sense that a behavior, once discovered, can often be exercised reliably once its dependent conditions are understood. This enables mitigations to be formulated with confidence.

However, if an adversary can find a behavior that the system owner is unaware of and that can be used to adversary's advantage, in many cases the adversary can cause this behavior to occur with 100% reliability. For example, a 40-bit input has over one trillion possible values. If precisely one of those leads to an exploitable vulnerability, it is incorrect to conclude that the system has one in a trillion chance of being exploited. If an adversary knows about this vulnerability and has direct control over the input, that adversary can exploit it with 100% reliability, because after that vulnerable input is found once, it can be deterministically reused as many times as the adversary desires.

4.3 Implications and Observations from Software History and Characteristics

The history of software development and the characteristics of that software lead to several significant implications. These implications inform us about the nature of the software understanding gap and what must be done to address it.

4.3.1 The Surprising Limitations of Testing

Time and time again, when software systems exhibit unexpected behavior, many intuitively respond by suggesting that "More testing is needed." The limitations of testing are surprising and not widely appreciated. The limitations are a direct result of the combination of characteristics described above.

The deterministic nature of software systems suggests that testing is valuable because the knowledge gained is not typically random or ephemeral. Testing validates that the software can do the things it was designed to do.

The discrete nature of software means that the tests run on software tell us nothing about the behaviors that might be exhibited by other test cases that weren't run. One solution to this, when the system is simple enough, is to exhaustively test every possible combination of inputs. This may be practical when the system is static and has no memory of past inputs, but as most software systems are dynamical, exhaustive testing must enumerate every possible combination of inputs over repeated interactions. Add to this the fact that modern software systems are staggeringly complex. The result of this collection of characteristics is that the number of possible states for even a modest program is typically many, many orders of magnitude larger than the number of atoms in the universe.

Consequently, it is impossible to exhaustively test all possible states of nearly all modern software systems. Non-exhaustive software testing is essentially a sampling method of discrete behaviors that explores an infinitesimal fraction of the total space. Although testing reveals *some* of what the software does, it cannot confidently determine what it *might* do. Testing cannot legitimately be used to interpolate or extrapolate to behaviors not specifically tested. A quote provided earlier (see Section 4.2.1) is worth repeating:

> *Since software lacks the continuity of physical systems that allow inferences to be drawn from one sample execution about neighboring points, testing says little or nothing about the cases that were not exercised.*[85]

Given the popularity of testing, these limitations may come as a surprise to many. To say that something is "secure" or "reliable" based on non-exhaustive testing is to misunderstand the nature of software and of testing and to draw unsupported conclusions. As a result, testing, despite its intuitive appeal, is a poor substitute for software understanding.

4.3.2 The Illusion of Predictability

As described above, great care is taken in the design of computing systems to ensure that they are deterministic, except in very limited cases. Consequently, given the same inputs, software will produce the same outputs, and will do this every time. This leads to repeatability—if you enter the correct password 100 times, 1000 times, or 10 million times, the computer system should permit you to log in every time.

But although repeatability is related to predictability, it is not the same thing. Repeatability involves *known* behavior and predictability includes *unknown or potential* behavior. Modern software systems are dynamical, discrete, non-linear, and complex. As Cliff Hooker describes it:

> *Non-linearities permit small differences in system state to be amplified into large differences in subsequent system trajectory.*[86]

There is a term commonly ascribed to phenomena that are sensitive to the tiniest changes in initial conditions, or that have seemingly random and unpredictable behavior, even though the system follows precise rules—this term is *chaos*.[87] Chaotic systems can exhibit emergent behavior that testing-based approaches cannot predict.

When one fails to appreciate the chaotic nature of software systems, there is a tendency to think that minor alterations to an input, perhaps by just changing a single bit, will have, at most, minor effects on the output. Although this may hold true in some instances, it is generally not true, and incorrectly assuming that it is true can lead to mission failure.

An unfortunate illustration of this is the unintended acceleration problem of certain Toyota vehicles. In the early 2000s, Toyota began using a software-controlled throttle system. In the years that followed, the National Highway Traffic Safety Administration (NHTSA) said that it received more than 6,200 complaints involving sudden acceleration in Toyota vehicles, including 89 deaths and 57 injuries.[88] The NHTSA brought in the National Aeronautics and Space Administration (NASA) to do a ten-month, independent review of the software:

> *In conducting their report, NASA engineers evaluated the electronic circuitry in Toyota vehicles and analyzed more than 280,000 lines of software code for any potential flaws that could initiate an unintended acceleration incident. … NASA engineers found*

---

[85] National Research Council, *Software for Dependable Systems: Sufficient Evidence?* (National Academies Press, 2007).
[86] Cliff Hooker, "Introduction to Philosophy of Complex Systems," *Philosophy of Complex Systems*, (Elsevier BV, 2011).
[87] Stanford Encyclopedia of Philosophy, "Chaos," Oct 13, 2015, https://plato.stanford.edu/entries/chaos/.
[88] CBS News, "Toyota 'Unintended Acceleration' Has Killed 89," May 25, 2010, https://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/.

> *no electronic flaws in Toyota vehicles capable of producing the large throttle openings required to create dangerous high-speed unintended acceleration incidents.*[89]

Despite these initial findings, subsequent investigation pursuant to ongoing lawsuits discovered unexpected behavior in the software involving "task death"—a "task" is a software routine that runs continuously, but "task death" occurs when a flaw causes it to stop executing. The results were described as follows:

> *Memory corruption as little as one-bit flip can cause a task to die. This can happen by hardware single-event upsets—i.e., bit flip—or via one of the many software bugs, such as buffer overflows and race conditions, we identified in the code. There are tens of millions of combinations of untested task death, any of which could happen in any possible vehicle/software state … [which] can cause loss of throttle control by the driver—even as combustion continues to power the engine.*[90]

In other words, this software had unexpected, emergent behavior that resulted in unintended acceleration and death; and there was not just a single way of provoking that behavior, but tens of millions, all of which remained undiscovered during testing and independent analysis by NASA.

Why was this software put into operation in the first place, despite its potential to have life-threatening, unexpected behavior? Because of the illusion of predictability that leads to false confidence in software. This same false confidence has led NS&CI missions to critically rely on software despite the accelerating software understanding gap.

4.3.3 Complexity, Abstraction, and Unexpected Behavior

Why do we use the term "unexpected" when talking about problematic software behavior? Sometimes it is because behavior wasn't discovered during testing or because we fail to understand that emergent behavior can arise in complex systems. However, there is a third source of unexpected behavior which can be illustrated with the following quote from software developer Joshua Bloch:

> *I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in Chapter 5 of Programming Pearls contains a bug…The version of binary search that I wrote for the JDK contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so.*[91]

How can an error (a "bug") occur in software that has been proven correct? Understanding the answer requires a brief discussion of the concept of *abstraction*.

To produce systems with the exceptional complexity described above which are also able to function well enough to be adopted by society, software developers have had to learn to build software using many layers of abstraction, which is the process of generalizing away from concrete details to

---

[89] U.S. Department of Transportation, "U.S. Department of Transportation Releases Results From NHTSA-NASA Study of Unintended Acceleration in Toyota Vehicles," August 1, 2019.

[90] Junko Yoshida, "Toyota Case: Single Bit Flip That Killed." EE Times. October 25, 2013. https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/.

[91] Joshua Bloch, "Extra, Extra – Read All About It: Nearly All Binary Searches and Mergesorts are Broken", June 2006 Google Research Blog. https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/.

simplified concepts.[92] In computer systems, abstraction is used as a form of information hiding, where the information that is hidden presumably consists of irrelevant details. For example, the abstract idea of an "integer" can take on any value, hiding the reality that numbers in computers are represented by a small, fixed number of 1's and 0's (called "bit vectors") and therefore can only represent limited values. Abstractions are a foundational concept in computer science and the proliferation of them is largely responsible for fueling the upper curve of Figure 2.

The bug mentioned in the quote above arose because the code was proven correct *for the abstraction*, not for the reality of how computers use bit vectors. The binary search program failed to take into account the possibility that adding two bit-vector numbers together would produce a result that could not fit in the available result bit vector; the computation could have been structured to avoid this possibility, but wasn't, resulting in the error. The abstraction was imperfect.

In software, there is nearly always a gap between the abstraction and the reality that the abstraction is attempting to simplify. Joel Spolsky coined the term "leaky abstractions" to describe such situations, concluding that "All non-trivial abstractions, to some degree, are leaky."[93] When adding two bit-vectors overflows the bits available to hold the result, the abstraction "leaks."

Programmers using an abstraction can fail to anticipate the potential behavior of the reality hidden by the abstraction. This is an example of why the software behavior so often deserves the label "unexpected". Seasoned software developers can learn to correctly navigate this issue for many abstractions, but given the complexity of modern software systems, none can do it for all.

4.3.4 The Impossibility of Perfect, Universal Understanding: Undecidability

From the earliest days of computing, when Alan Turing introduced his idea of the Turing Machine, it was realized that there were limits to what computer algorithms would be able to do. Turing not only introduced the world to the idea of the programmable universal computing machine, but also to a key limitation of it. Turing laid the groundwork to prove that you cannot develop one Turing Machine capable of determining whether an arbitrary, second Turing Machine would halt on any input. In the 1950s this became known as the "Halting Problem" and many related proofs about the limitations of software to analyze other software were formulated. One of the most important was Rice's Theorem:

> *All non-trivial semantic properties of programs are undecidable. A semantic property is one about the program's behavior, unlike a syntactic property.*[94]

The term *undecidable* is a technical one indicating a special kind of impossibility. A decision problem is a yes/no question on input values; if that question is always answerable for all inputs, then it is *decidable*. If there is no algorithm that is guaranteed to always produce a yes or no answer regardless of input, then the problem is *undecidable*. That is, the question of decidability is one of *universal guaranteed success*. When a problem is undecidable, it's worth nothing that this does not mean that no answer will ever be produced—it may very well be that an algorithm can produce correct answers on many programs of interest, even though it cannot be guaranteed to work on all.

---

[92] Timothy Colburn; Gary Shute, "Abstraction in Computer Science," (*Minds and Machines*. 17 (2): 169–184, 2007) https://link.springer.com/article/10.1007/s11023-007-9061-7.
[93] Joel Spolsky, "The Law of Leaky Abstractions," Joel on Software, 2002, https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.
[94] https://en.wikipedia.org/wiki/Rice%27s_theorem Accessed Dec 6, 2023.

On first reading, Rice's Theorem would seem to state that you cannot create programs that answer questions about other programs at all. This is a very common misunderstanding. Fortunately, the goal of software understanding does not depend on a universal guarantee of success. Saying that something cannot *always* work does not mean that it will *never* work, and to narrow the software understanding gap in NS&CI missions, partial success and summarized information can provide significant benefit. Asking carefully chosen questions about a specific software program can be tractable. For example, by focusing on specific instances instead of universal generalities, it is possible to identify many non-terminating conditions, the Halting Problem notwithstanding.[95]

The implications of these theoretical results are that a "one-size-fits-all" automated software understanding capability cannot be created that is guaranteed to work perfectly over all software of interest. To proceed practically, software understanding capabilities may need to be tailored to the software under evaluation, guided at times by human insight, or free to provide partial results which can inform further investigation. (See Question #1 in Appendix B for more information.)

4.4 Conclusions and the Software Understanding Gap

Given the discussion above regarding the historical dynamics and characteristics of software, there is little surprise that a significant and growing software understanding gap would exist. Because of the generally deterministic nature of software, if an adversary can identify a mission-threatening behavior within the software understanding gap that the system owner is unaware of, then that behavior can often be triggered with 100% reliability. The massive interconnectivity of computing systems across the planet means that adversaries no longer must enter physical compounds to attack national security systems but can directly or indirectly probe critical systems from anywhere on the globe. Because of the software understanding gap, the third-party and open-source software supply chains open avenues of attack which lack adequate technical capabilities to mitigate. All told, problems arising from the software understanding gap result in global economic damages from cybercrime of trillions of dollars annually.

In NS&CI mission spaces, the need for reliability is paramount. Challenges of cost-effective, reliable software engineering in the 1980's and 1990's led to a key policy decision to embrace COTS systems and eschew GOTS when possible. At the time, given the relatively limited connectivity among systems and the limited awareness of software vulnerabilities and attack opportunities, this decision made sense and ushered in revolutionary NS&CI capabilities by embracing commercial and third-party software. Global inter-connectivity and widespread awareness of how to "hack" software have created a threat landscape unlike any seen in history, not only from nation-state actors, but also from organized crime and even just curious teenagers. Ken Thompson demonstrated that reviewing source code alone can never lead to high levels of confidence in software. The exploding number of CVEs demonstrates that unexpected behavior in software is not a rare phenomenon, but altogether too common. Collectively, these trends have undermined the suppositions that justified our historical approaches to risk calculus for software in NS&CI missions. These trends are now motivating the need for new approaches able to address the present software understanding gap. Each instance of a CVE in software used in NS&CI missions stands as evidence that mission owners lack the necessary capabilities to prevent putting vulnerable software into service in the first place. The software understanding gap now threatens NS&CI missions in ways that it never did in previous decades.

---

[95] PolySpace Technologies, https://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/POLYSPACE/polyspace-daedalus.htm

5 Software Understanding Case Studies

To illustrate both the need and the challenge of understanding software, this section presents four detailed case studies. Each case study discusses an example of unexpected behavior that was accidentally or intentionally inserted into software, sometimes innocently and sometimes maliciously. The case studies involve a variety of actors with a variety of motivations.

5.1 Volkswagen Emissions Scandal

In 2017, Volkswagen pled guilty to criminal charges and paid $4.3 billion dollars in criminal and civil penalties for cheating on emissions tests and selling diesel vehicles that produced emissions up to 40 times higher than legal emissions standards.[96] To accomplish this, Volkswagen inserted changes in their vehicle control software so that it would follow emissions limits prescribed by regulations during emissions testing procedures, but ignore these limits during real-world driving conditions, delivering maximum performance to the driver. This was accomplished by secretly modifying the proprietary vehicle control software to precisely monitor certain vehicle characteristics (e.g., steering wheel angle position, speed, time driven). These modifications were designed to recognize whether the vehicle was experiencing the conditions specified by various governments' emissions tests or the conditions that would occur during normal driving activities. Modern automobiles contain hundreds of electronic control units (ECUs) and millions of lines of code—this software modification was made to a small portion of just one of these.[97] These software modifications, designed to evade testing, were installed on nearly 60,000 vehicles sold from 2009 through 2016.

The behavior was initially discovered when a 2014 International Council on Clean Transportation (ICCT) study tested emissions in ordinary driving conditions instead of a standard emissions testing environment.[98] A paper providing a detailed analysis of the software modification noted:

> *As software control becomes a pervasive feature of complex systems, regulators in the automotive domain (as well as many others) will be faced with certifying software systems whose manufacturers have an immense financial incentive to cheat.*[99]

This example involves automobile emissions, various governments' emissions testing regimens, and the financial incentives of the former to cheat the latter. This is not the only example of various entities placing malicious modifications in software for gain or other personal motives.[100,101,102]

---

[96] U.S. Department of Justice, "Volkswagen AG Agrees to Plead Guilty and Pay $4.3 Billion in Criminal and Civil Penalties; Six Volkswagen Executives and Employees are Indicted in Connection with Conspiracy to Cheat U.S. Emissions Tests", 2017, https://www.justice.gov/opa/pr/volkswagen-ag-agrees-plead-guilty-and-pay-43-billion-criminal-and-civil-penalties-six.
[97] Robert Charette, "How Software is Eating the Car," IEEE Spectrum, June 2021, https://spectrum.ieee.org/software-eating-car.
[98] Harry Kretchmer, The man who discovered the Volkswagen emissions scandal," BBC Radio, October 2015, https://www.bbc.com/news/business-34519184.
[99] M. Contag "How They Did It: An Analysis of Emission Defeat Devices in Modern Automobiles," 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 2017) pp. 231-250, doi: 10.1109/SP.2017.66, https://ieeexplore.ieee.org/document/7958580.
[100] Adam Haertle, "About three people who hacked a real train – or even 30 trains," May 12, 2023, https://zaufanatrzeciastrona.pl/post/o-trzech-takich-co-zhakowali-prawdziwy-pociag-a-nawet-30-pociagow/ (English translation available at https://badcyber.com/dieselgate-but-for-trains-some-heavyweight-hardware-hacking/).
[101] Dan Goodin, "Contractor admits planting logic bombs in his software to ensure he'd get new work," December 19, 2019, https://arstechnica.com/tech-policy/2019/12/contractor-admits-planting-logic-bombs-in-his-software-to-ensure-hed-get-new-work/.
[102] "Ex-Navy contractors gets year," *The Washington Times*, April 5, 2007, https://www.washingtontimes.com/news/2007/apr/5/20070405-114823-5425r/.

5.1.1 Implications for NS&CI Missions

Let us consider a national security version of this case study. Instead of an automobile manufacturer, consider a hypothetical vendor of a software-driven device used in a national security mission. Every such vendor is faced with cost, schedule, and performance constraints that are in tension. Consequently, there is a financial incentive for such vendors to minimize cost and avoid scheduling delays (both of which are highly visible to the mission owner) at the expense of some functionality, which may not be noticed in testing and could remain undiscovered by the mission owner for many years. Government mission owners establish rigorous test and evaluation regimens to ensure that the system performs as required, but such regimens cannot be exhaustive due to the characteristics of software discussion in Section 4.2. Consequently, even rigorous testing regimens are insufficient for high consequence systems.

Let us examine this scenario—intentional evasion of software testing—using the language of mission, mission questions, and mission success or failure. Many national security missions call for a high-degree of confidence that the system under evaluation does not have some modification designed to "cheat the test", as in the Volkswagen example. Let us suppose that it is essential to the "mission" of a particular national security organization to determine whether a given family of devices acquired for use in that mission will follow a particular standard regardless of operating conditions. *Mission success* would then be correctly determining either that the devices in question cannot violate the standard, or that they could. *Mission failure* would be failing to correctly determine this. Informed by the fraud involved in the Volkswagen case, one could refine *mission success* to be correctly determining that the devices cannot violate the standard, *even if the manufacturer is adversarial and attempting to evade detection*.

To achieve mission success for this part of its mission, this national security mission organization would ideally ask the *mission question*:

> *"Can this family of devices violate the standard under any condition?"*

In cyber-physical systems such as vehicles, this can be a challenging question to answer, involving many domains of expertise. In modern software-controlled systems, this question further involves understanding the potential behavior of the software. For the vast majority of systems, this mission question cannot be answered today. Given the way most software is developed, the technical capabilities do not exist to analyze software systems rigorously enough to gather the technical evidence needed to answer this question with high confidence in the face of adversarial attempts to evade detection.

As an alternative approach to the ideal mission question, most organizations turn to testing to assess software-based systems relative to their mission needs; given current capabilities for analyzing software-based systems, these organizations have little choice. A far simpler mission question would be:

> *"Does this device violate the standard during this precisely defined test?"*

This question can be answered in a simple manner, by merely testing the device and observing whether its behavior deviates from the standard during the precisely defined test. However, as discussed in detail in Section 3.3.1 and 4.3.1, testing can only reveal the specific behaviors elicited by a given test; it cannot validly be used to rule out other, undesirable behaviors which weren't

examined by the tests.[103] A completely different approach and set of capabilities are required to answer the former, ideal mission question as opposed to the latter, simpler one. Testing is insufficient to achieve high confidence regarding the former, ideal mission question.

No current organization should be faulted for not assessing software relative to the ideal mission question since the capabilities to do so do not yet exist.

In years past, limitations of physical, mechanical systems meant that sophisticated testing defeat techniques like the one in the Volkswagen case were almost unimaginable. With the introduction of modern software into vehicles and numerous national security systems, covert triggers like the one designed by Volkswagen can be amazingly sophisticated yet fit into an infinitesimal space, opaque and inscrutable to traditional testing approaches.

From the perspective of the challenge of understanding potential behavior in software, this case study illustrates several points:

- Aspects of the software's behavior were intentionally designed to evade detection through testing.
- What is needed to detect such malicious behavior in software-based systems are tools that can analyze software directly to provide better understanding of its *possible* behavior without having to physically test it in all possible conditions and configurations.
- National security mission concerns regarding software go far beyond currently available software analysis tools that look for malware signatures, software crashes, or software vulnerabilities.
- Attestation-based approaches are inadequate in cases such as this one. Not only did Volkswagen deliberately seek to evade detection, the history of the "Dieselgate" scandal is littered with false statements, evasion, and a lack of transparency on Volkswagen's part despite claims to the contrary.[104]

5.2 SolarWinds

In March of 2020, the data, network, and systems of thousands of public and private organizations were compromised when a supply chain attack inserted a backdoor into an otherwise routine update to the popular Orion monitoring and management software platform, produced by the company SolarWinds to simplify IT administration.[105] More than 18,000 SolarWinds customers installed the malicious update, including nine USG federal agencies.[106] The malicious backdoor enabled full, privileged access to the software via remote internet domains. The malicious aspects of the update were not discovered until December of 2020, giving the attackers at least 9 months of hidden, unfettered access to customers' systems. The USG eventually attributed the compromise to the Russian Foreign Intelligence Service.[107]

---

[103] Unless the tests are exhaustive, but exhaustively testing all possible software states is typically infeasible.

[104] "VW's Crisis Strategy: Forward, Reverse, U-Turn," *New York Times*, February 26, 2016. https://www.nytimes.com/2016/02/28/business/international/vws-crisis-strategy-forward-reverse-u-turn.html.

[105] "The Untold Story of the Boldest Supply-Chain Hack Ever," *Wired Magazine*, May 2, 2023, https://www.wired.com/story/the-untold-story-of-solarwinds-the-boldest-supply-chain-hack-ever/.

[106] U.S. Government Accountability Office, "SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response," April 22, 2021, https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic.

[107] "Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations," CISA Advisory April 15, 2021, https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a.

A technical report describing the attack highlights revealed that, unlike the Volkswagen example above, SolarWinds' source code was not modified to contain the malicious changes; instead, the malicious software was covertly inserted into the Orion platform's executable program during the build process and prior to the code being digitally signed.[108] SolarWinds executives and Orion developers were all unaware of the modifications. As a result, the backdoor was present in digitally signed updates supplied by Solar Winds, a company whose software was trusted for years to monitor and manage many important networks throughout government and society.

In the process of remaining undetected for 9 months from 18,000 customers and SolarWinds itself, the malicious code inserted into Orion was undoubtedly subjected to rigorous testing not only by the developer, but also by many thousands of users through their own independent tests, numerous scans with malware detection tools and other security products, and daily use for many months. How was the malicious code able to hide from such extensive scrutiny? Subsequent detailed forensics reports revealed that it was specifically designed to elude detection from automated testing and to blend in with legitimate activity whenever possible. The malicious code would lie dormant for a randomized time between 12 and 14 days after installation before activating. Once activated, it would scan the system for known software and malware analysis tools and attempt to disable them. Fixed strings were obfuscated, command and control characteristics were randomized, and more. Collectively, the full set of anti-forensic techniques proved effective at frustrating available methods of detection for months.[109,110]

5.2.1 Implications for NS&CI Missions

Let us consider this case study using the language of mission, mission questions, and mission success or failure. Let us suppose that any customer of a cybersecurity product should have, as part of their "mission," to ensure that their cybersecurity products never permit a network connection from a remote internet domain to access privileged functions. *Mission success* would then be determining with high confidence that no such network connection is possible or detecting that one is possible so that either the connection can be mitigated or the product rejected for use. *Mission failure* in this case would be failing to detect that such a connection is in fact possible.

To achieve mission success, the customer of a cybersecurity product would ideally ask the *mission question*:

> *"Could this software ever permit a network connection from a remote internet domain to access privileged functions?"*

Gathering technical evidence for this mission question requires analyzing the potential behavior of the software. One far simpler mission question that was likely asked instead by many customers is:

> *"Is this software update digitally signed by the SolarWinds certificate?"*

Many tools exist today to verify a digital certificate and using them is considered a best practice. However, verifying that a digital signature is present is not the same as determining that the

---

[108] CrowdStrike Intelligence Team, "SUNSPOT: An Implant in the Build Process," January 11, 2021, https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/.

[109] Mandiant, "Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor," Mandiant, 2020, https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.

[110] "Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations," CISA Advisory, 2021, https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a.

behavior of the software bearing the signature is trustworthy. Using the digital signature as a proxy for genuine understanding of the software's potential behavior led to mission failure in this case for many organizations.

The case study can also be considered from the perspective of a vendor of such a product, not just the users. Let us suppose that any such vendor should have as part of their "mission" to sell a product that is not compromised by a clandestine backdoor that could permit a network connection from a remote internet domain to access privileged functions. SolarWinds' statements suggest that the company was not able to succeed at this part of their mission with the testing approaches they and so many other companies employ as a standard part of the development process. Once the Orion platform backdoor was discovered, SolarWinds was under intense public scrutiny and highly motivated to ensure that no similar backdoor existed in their other products.

After the discovery, SolarWinds reported, "We have scanned the code of all our software products for markers similar to those used in the attack on our Orion Platform products, and we have found no evidence that our other products contain those markers."[111] It is both ironic and telling that SolarWinds reported results for the question "Do my other software products contain markers similar to this known malicious code?" instead of the ideal, more rigorous question "Are my other software products compromised in any way?" The likely reason is that neither SolarWinds, nor the customer, nor any other cybersecurity developer currently has the technical capabilities necessary to answer this ideal question.

From the perspective of the challenge of understanding potential behavior in software, this case study illustrates several points:

- Unlike the Volkswagen case above, neither SolarWinds executives nor their software developers were complicit in adding the undesirable behavior to the software.
- Attestation-based approaches are inadequate in cases such as this one. SolarWinds decision makers believed their software to be uncompromised.
- Existing software analysis capabilities are inadequate to answer the ideal mission question, "Is this software compromised with a backdoor?" Answering the proxy questions and relying on existing capabilities led to mission failure.
- Even if perfect, rigorous analysis had been conducted on Orion's source code, the compromise would never have been found. The compromise occurred *during* the build process and was only present in the actual executable. This underscores the need to analyze software artifacts that will be run rather than just source code.
- Despite extensive testing and use by the vendor and the numerous customers compromised, the malicious functionality remained hidden, demonstrating the limitations of testing (see Section 4.3.1).
- The number of ways that software may be malicious is bounded only by human creativity. When faced with the question of whether other products have malicious backdoors, SolarWinds' public response suggests an understanding that the particular markers of this malicious backdoor are not the only possible way a backdoor could be present.
- The lack of technical capabilities to analyze vital software and gather technical evidence to adequately answer critical mission questions led to mission catastrophe—remote

[111] Catalin Cimpanu "SolarWinds said no other products were compromised in recent hack," ZDNet, 2020, https://www.zdnet.com/article/solarwinds-said-no-other-products-were-compromised-in-recent-hack/.

administrative access by hostile foreign intelligence to thousands of customers and multiple U.S. government agencies' internal networks and systems.

5.3 Log4j

Log4j is a popular open-source Java library for logging error messages. In July 2013, a new patch was submitted to the Log4j library, adding a new feature that was believed to be useful to a wide range of developers. The changelog comment was "It would be really convenient to support JNDI resource lookup in the configuration." The change was accepted by the Log4j project in less than 24 hours.[112] In December 2021, NIST's NVD first reported this as a vulnerability with a criticality score of 10.0 (out of 10), due in part to the ease with which the exploit could be executed.[113,114] The vulnerability (called "Log4Shell") permits a hostile user with influence over the error message being logged to be able to perform remote code execution (RCE), one of the worst kinds of vulnerabilities because it frequently permits an attacker complete control over the application in question. By the time the vulnerability was discovered, the library was in use by missions of computers worldwide running online services, including various governments.[115] This would prove to be the top vulnerability exploited by Chinese state-sponsored cyber actors for the next year.[116]

The chain of events in the source code that led to the vulnerability involves the very particular interaction of over a dozen functions.[117] Stephen Magill reported that "in one application, there are 30 levels of dependencies separating the application from Log4j," making human analysis extremely challenging. In the same article, he also reported that there were approximately 70,000 applications directly dependent on Log4j, with approximately 175,000 indirectly dependent on it.[118]

The vulnerability was present but unknown to the public for over 8.5 years. This is not the only vulnerability to exist unknown to the public for roughly a decade. A vulnerability discovered in 2021 in the privileged Linux "sudo" command existed for over a decade before it was discovered.[119] A vulnerability discovered in 2022 in the privileged Linux tool "Polkit" lay undisclosed for 12 years.[120]

5.3.1 Implications for NS&CI Missions

Let us consider this case study using the language of mission, mission questions, and mission success or failure. As mentioned above, numerous government agencies were using software containing the Log4j vulnerability when it was discovered. Let us suppose that part of the "mission" of these agencies is to ensure that the software they use does not enable user input to cause remote code to be downloaded and executed. *Mission success* would then be correctly determining

---

[112] See Log4j issue report at https://issues.apache.org/jira/browse/LOG4J2-313 (accessed November 1, 2023)
[113] NIST, "CVE-2021-44228 Detail," December 10, 2021, https://nvd.nist.gov/vuln/detail/CVE-2021-44228.
[114] https://en.wikipedia.org/wiki/Log4Shell
[115] National Cyber Security Centre, "Log4j vulnerability—what everyone needs to know," December 14, 2021, https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know.
[116] CISA, "Top CVEs Actively Exploited by People's Republic of China State-Sponsored Cyber Actors," October 6, 2022, https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-279a.
[117] Hardik Shah, Sean Gallagher, "Inside the code: How the Log4Shell exploit works," December 17, 2021, https://news.sophos.com/en-us/2021/12/17/inside-the-code-how-the-log4shell-exploit-works/.
[118] Stephen Magill, "What Log4J teaches us about the Software Supply Chain," ThinkstScapes Quarterly, Q2 2022, https://www.youtube.com/watch?v=SAlylyyiy9U.
[119] Assaf Morag, "CVE-2021-3156 sudo Vulnerability Allows Root Privileges," Aqua Blog, 2021, https://blog.aquasec.com/cve-2021-3156-sudo-vulnerability-allows-root-privileges.
[120] ArsTechnica, "A bug lurking for 12 years gives attackers root on most major Linux distros," January 25, 2022, https://arstechnica.com/information-technology/2022/01/a-bug-lurking-for-12-years-gives-attackers-root-on-every-major-linux-distro/.

with high confidence whether any given software package does or does not have such a possible behavior. *Mission failure* would be failing to realize that software placed into service does have such a vulnerability.

To achieve mission success, a government agency would ideally ask the *mission question*:

> *"Could this software permit user input to cause code to be downloaded and executed?"*

In this case, it is unclear what mission question, if any, the agencies using the software were in fact asking or analyzing the software to answer. Lack of capability to understand the software's potential behavior well enough to answer the ideal mission question prior to putting it into service resulted in mission failure for this part of those agencies' mission.

From a software understanding perspective, this case study illustrates several points:

- Attestation-based approaches were inadequate. The developer added what was understood to be a beneficial feature. Users who passed user-controlled data to the library failed to realize the ramifications.
- Serious vulnerabilities can go undetected for many years, demonstrating the current lack of adequate technical capabilities.
- The extended time between introduction of the vulnerability and public discovery creates a lengthy window of operational advantage to the potential adversary. Any adversary who discovers the vulnerability may enjoy many years of unilateral exploitation for their own purposes. It should be expected that similarly critical vulnerabilities exist right now in software that is in widespread use, yet which remain unknown to the public but may be known to hostile actors—a situation of unknown and unbounded risk.
- Modern software is incredibly complex. The Log4j code is open source and available for scrutiny by anyone. Many of the vulnerable applications that used it were also open source. Yet the complexity and other characteristics of modern software make it increasingly difficult for humans to adequately reason through potential behavior (see Section 4.2). This suggests that automated tools are necessary to help mission owners adequately analyze modern software to identify potential mission risk and guide mitigations and other risk decisions.
- Section 3.3.7 discussed the utility of an SBOM for identifying the components of software and to trace dependencies. Once a vulnerability like the one in Log4j was discovered, SBOMs (in cases in which they were available) are one tool that could be used to identify which applications could also be affected. It is worth noting, however, that SBOMs in no way help detect novel vulnerabilities.
- Beyond RCE, there are a wide range of negative effects that end users, companies, and government agencies would like to ensure are not present in the software they use. For example, it would be important to mission success to know that an adversarial user input was unable to do any of the following: delete sensitive files, expose restricted information, trigger mission-critical actions without authorization, prevent authorized mission-critical actions from occurring, digital sign counterfeit communications to appear authentic, and many more. Automated technical capabilities do not currently exist to assist in analyzing critical software to help mission owners understand whether such potential behavior may be present.

5.4 Node-ipc

The "node-ipc" package is a popular open-source package for local and remote inter-process communication with over 4 million monthly downloads. In March 2022, in response to Russia's invasion of Ukraine, the author of "node-ipc" inserted code into the released version to mount a destructive supply-chain attack against Russian computers using the package. To target the destructive behavior toward a single organization, the author inserted a *trigger* to ensure that only certain systems would be affected. The changes to "nope-ipc" geo-located the computer on which the software was running; a location of Belarus or Russia would trigger the destructive behavior which would overwrite all files on the system volume with a heart character.[121,122]

5.4.1 Implications for NS&CI Missions

Let us consider this case study using the language of mission, mission questions, and mission success or failure. Although the targeted country in this scenario was Russia, there is nothing technically that limits the scenario to any specific country. As many countries who have outspoken critics depend on open-source software, let us consider an arbitrary country engaged in an international conflict and how it may be vulnerable to an attack like this one. Let us suppose that part of the "mission" of any nation so engaged in a conflict is to not leave itself exposed to a single, rogue individual being able to mount a destructive cyber-attack against it. During times of conflict, it may be discovered that enemy actors are exploiting existing vulnerabilities, demanding immediate updates to new software versions to close active avenues of exploitation. In such cases, the new versions need to be vetted rapidly to ensure that mission threatening behavior isn't present in the update. This presents a dilemma to mission owners: updating software quickly is necessary for closing the window on enemy cyber-attacks, but the faster software is updated, the less time there is to vet the update for new malicious behavior.

Mission success would be analyzing existing software and updates rapidly, at mission-speeds, to accurately determine with high-confidence whether any potentially destructive behavior is present in the software. *Mission failure* would be if software were unknowingly put into use that had such destructive behavior.

Unfortunately, most software has "destructive" behaviors that are benign when invoked by an authorized user—in many cases these are necessary features, so simply knowing that software is capable of destructive behavior is not enough to determine it is malicious. When these same behaviors are triggered automatically by unexpected conditions, they may be indications of a malicious supply chain attack. There are a wide range of triggers that go beyond geo-location that may be of interest to a scenario like this one. For example, it would be important to mission success to know prior to installing it that the sensitive or irreversible actions of a software package could potentially be triggered by any of the following: time of day or date,[123] proximity to a particular Wi-Fi SSID, a "magic string" in a communication message, a particular GPS "box," time since the system power on or application started, language pack, time zone, computer domain name or IP address, etc.

---

[121] Lucian Constantin, "Developer sabotages own npm module prompting open-source supply chain security questions," March 19, 2022, https://www.csoonline.com/article/572327/developer-sabotages-own-npm-module-prompting-open-source-supply-chain-security-questions.html.

[122] Chris Thompson, Protestware – How node-pic turned into malware (LunaSec, 2022).

[123] The CIH (Chernobyl) virus released in 1998 was set to deliver its malicious payload on April 26, 2999, one of the anniversaries of the Chernobyl nuclear incident. For more information, see https://en.wikipedia.org/wiki/CIH_(computer_virus).

Achieving mission success in the face of such triggers governing the overwriting of system files (the effect of the malicious "node-ipc" update), mission owners would ideally ask the *mission question*:

> *"What non-user-initiated conditions could cause system files to be overwritten?"*

This mission question is relevant to detecting certain ransomware attack vectors as well. The tools necessary to enable mission owners to answer this mission question against the full set of mission-critical software do not exist. From a software understanding perspective, this case study illustrates several points:

- The update was a deliberate supply-chain attack made by *a single individual* as a form of protest.
- Dependence on open-source software anywhere within military operations may enable a single, private individual armed only with a keyboard to produce a military-like effect through a supply chain attack.
- During an on-going conflict, answers to questions about the potential behavior of software are needed immediately. This suggests that automated tools are necessary to help mission owners meet operational deadlines while reliably and robustly analyzing software for potentially destructive behaviors of interest.
- Attestation-based approaches were inadequate. An individual willing to engage in a destructive supply-chain attack would not be expected to answer truthfully to any attestation statements.
- Testing approaches are also inadequate because the trigger for causing malicious behavior may be carefully designed to never activate during testing but to activate immediately in the field/time of conflict.

5.5 Conclusion

Our case studies show that unexpected behavior can arise in a variety of ways: both from the intention of the software developer (Volkswagen and Node-ipc) and without the developer's intention (SolarWinds Orion and Log4j); both from malicious motivations (Volkswagen, SolarWinds Orion, and Node-ipc) and from well-intentioned features meant to be of benefit (Log4j).

For the "node-ipc" case study, no public data is available on how many Russian and Belarus systems were damaged by the supply chain attack. For the other three examples, all were discovered after the fact, when the software was already in use.[124] In each of these cases, once discovered, the behavior was considered unacceptable. Yet in none did the end user detect the undesirable behavior before putting the software into use. Doing so is vital to mission success but is not currently an option due to lack of technical software understanding tools needed to analyze software for potential behavior. Because of this lack of technical capability, in these examples, mission owners did not ask the ideal mission question needed for mission success but instead were forced to ask weaker mission questions that matched the capabilities at hand. Mission owners should not be faulted for doing so.

In all case studies, attestation approaches were inadequate because of various reasons: in the Volkswagen and node-ipc cases, the software authors intended to keep the behavior from discovery;

---

[124] No public data is available on how many Russian and Belarus systems were damaged by the "node-ipc" attack. Given the nature of the trigger, it is possible that simple testing on computers in Russia or Belarus would have detected the malicious behavior before putting the software into operation. Coupling the destructive action with the types of triggers present in the SolarWinds case would have made such detection far less likely.

in the SolarWinds Orion case, they were unaware of the behavior; and in the Log4j case, they didn't realize the negative ramifications of the feature they knowingly added.

In all case studies, a key factor that led to mission failure was that mission owners did not receive answers to key mission questions based on a technical analysis of the software itself. To avoid the mission failures described above, mission owners would have needed cost-effective, rapid options to build a technical evidence package of potential mission-threatening behavior derived from analysis of the software used in mission systems. As illustrated by the SolarWinds case study and Ken Thompson's example (see Section 4.1.2), such options must analyze the final executable and not just the source code.

These case studies represent just the tip of the iceberg. There are tens of thousands more examples of software having unexpected, mission-threatening behavior being placed into service without proper understanding of the software's potential behavior. We should expect that there are dozens, if not hundreds, of other software compromise events in the United States that are in progress right now that remain undiscovered. At the same time, these case studies illustrate and motivate the types of rigorous, reliable, and rapid automation tools needed to adequately understand software with respect to a wide range of mission questions before that software is placed into service.

6 Options for Addressing the Software Understanding Gap

The examples and case studies of the previous section illustrate the real-world challenges and consequences that arise from unexpected behavior in software. These examples represent billions of dollars of damage, years of vulnerability to malicious actors, significant loss of trust in government institutions' ability to protect the nation's data, and even loss of life.

How should the nation address the issue of mission risk from unexpected behavior in software? The characteristics of software described in Section 4 show that software understanding is a difficult endeavor. The scale of the problem only exacerbates the challenge. Looking back at the chronology presented in Section 4, there is no point in history where the nation came to a carefully reasoned decision and plan about how to address this software understanding gap.

This section discusses three options for addressing the software understanding gap. These options are neither exhaustive nor mutually exclusive. The options are:

1. Accept the software understanding gap as inevitable,
2. Redouble efforts in software understanding alternatives, or
3. Pursue automated technical solutions to software understanding.

The sections below will discuss each of these in turn.

6.1 Accept the Software Understanding Gap as Inevitable

Argument in Favor. The argument in favor of this option is simple. Unexpected behavior in software should be expected. Given the characteristics of software discussed in Section 4, the downsides of using software, regrettable as they are, are part and parcel with the benefits. Just as occasional illness in humans is to be expected, problems stemming from the use of software are inevitable. And just as vehicles have transformed society but bring with them a cost from accidents measured in both money and human life, driving the costs to zero is not possible without removing much of the value. The widespread proliferation of software in recent decades has revolutionized

society and government and will continue to do so despite unexpected behavior arising from the software understanding gap. Reducing the gap to zero is neither necessary nor feasible.

Furthermore, market pressures are already coming to bear on software developers to construct software with better security, better privacy, fewer vulnerabilities, etc. This market pressure acts as a counteraction to the growing software development curve of Figure 2, squeezing the software understanding gap from above. The software understanding gap is not a problem the United States government should try to solve.

Critique of the Argument. It is true that unexpected behavior in software has occurred for years and does not appear to have slowed the widespread adoption of software. Society has benefitted greatly from widespread use of software while avoiding widespread calamity. However, more and more segments of society are coming to critically rely on software, including NS&CI missions. The trendlines of unexpected behavior in software continue to rise, as do concerns about growing consequences. When the deadly Therac-25 software problems were discovered, those radiation therapy machines represented a very tiny fraction of overall cancer radiation treatments that year. As NS&CI missions continue to be transformed by software, the entire foundation of society is increasingly in the hands of software. It cannot be expected that the consequences will remain acceptable.

Furthermore, recent years have shown a rise in supply chain attacks through software, such as those described in Section 5.2 (SolarWinds) and Section 5.4 (Node-ipc). Given the globalization in software development, adversaries hostile to the United States can be expected to continue to take advantage of our growing dependence on commercial and open-source software, learning to hide better within the growing software understanding gap.

If it were the case that closing the software understanding gap could happen quickly once the consequences arising from it become unacceptable, then delay could be a viable option. If, on the other hand, addressing the software understanding gap requires decadal investments to advance foundational research and develop scalable, automated tools that can reason about software's potential behavior, then delay will only make the growing gap more difficult to address once we start. As an analogy, national efforts to cure cancer have been ongoing since President Nixon signed the National Cancer Act in 1971.[125] After more than 50 years, the work continues. Addressing the software understanding gap may require similar timelines.

Waiting until financial and human life losses have accelerated to unacceptable levels before starting to address the software understanding gap is to invite disaster.

6.2 Redouble Efforts in Software Understanding Alternatives

Argument in Favor. As discussed in Section 3.3, there are many alternatives available to address concerns arising from unexpected behavior in software. These include, but are not limited to, testing, the use of digital signatures, dynamic system monitoring, reverse engineering, mitigations, SBOMs. These approaches are tried and true and were each introduced specifically to address key problems arising from software. Although they don't address the software understanding gap directly, they indirectly address many of the consequences. However, to date these alternatives have not yet been rigorously pursued in a consistent, all-of-government and all-of-society fashion. Recent federal government executive orders and strategy documents are expanding the groundwork to

[125] USG. National Cancer Act of 1971. (1971) https://www.congress.gov/92/statute/STATUTE-85/STATUTE-85-Pg778-3.pdf

pursue such alternatives in earnest. We should redouble efforts to induce software producers and consumers, especially in NS&CI mission spaces, to adopt these measures.

There are two alternatives that are effective already and thus warrant a redoubling of effort. First, the use of testing should be dramatically expanded. Testing is a precise and powerful method that can identify errors in software before it is put in use. Although tests cannot generally be exhaustive (see Section 4.2.4), it is not strictly necessary to be exhaustive from a mission success perspective. In addition, testing provides an important feedback mechanism on developers that improves software overall, well beyond what the tests themselves may cover. As Turing Award winner Tony Hoare put it in a 2009 retrospective article:[126]

> *One thing I got spectacularly wrong. I could see that programs were getting larger, and I thought that testing would be an increasingly ineffective way of removing errors from them. I did not realize that the success of tests is that they test the programmer, not the program. Rigorous testing regimes rapidly persuade error-prone programmers (like me) to remove themselves from the profession. Failure in test immediately punishes any lapse in programming concentration, and (just as important) the failure count enables implementers to resist management pressure for premature delivery of unreliable code. The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct.*

The second alternative worthy of a redoubled effort is manual reverse engineering to answer mission questions. Manual software analysis through reverse engineering (RE) has been an effective, tried-and-true approach to gain software understanding for decades. The nation should redouble efforts to grow the pool of human RE SMEs to meet mission needs. This would enable NS&CI mission owners to perform effective software analysis on a larger range of systems than we can with our current, limited RE expertise pool, shrinking the software understanding gap.

Critique of the Argument. It is true that the alternatives discussed in Section 3.3 have undoubtedly had a positive effect on addressing problems arising from software; additional investment would likely be beneficial. However, despite the benefits, it must be acknowledged that these alternatives are not nearly sufficient for closing the software understanding gap. Even if all the proposed alternatives were put in place, unacceptable NS&CI mission failure could still occur because of unexpected software behavior, especially behavior resulting from supply chain attacks. Many of the alternatives are forms of attestation, which often fails, as the case studies in Section 5 illustrate. Regarding testing, Tony Hoare also wrote in the same retrospective article quoted above:

> *A hacker exploits vulnerabilities in code that no reasonable test strategy could ever remove…The only way to reach these vulnerabilities is by automatic analysis of the text of the program itself.*

Section 4.3.1 explains the reasons why software testing cannot rule out behavior that may imperil the mission; testing therefore should not be the primary source of confidence in missions that depend on software.

Although manual RE was the mainstay method for software understanding for decades, in the face of massive expansion of software adoption into so many critical systems, the growing complexity of

[126] C.A.R. Hoare, “Retrospective: An Axiomatic Basis for Computer Programming,” Communications of the ACM, Oct 2009, https://dl.acm.org/doi/pdf/10.1145/1562764.1562779.

software, and the accelerating pace of software creation and updates, manual RE is already unable to keep up with mission needs. Furthermore, the quality and speed of human analysis also declines as the problem increases in complexity, raising costs while simultaneously weakening results. The software development boom didn't just occur because there were more software developers, but because the production tools they leveraged advanced dramatically, fueling interoperability, reuse, and lowering cost. Absent a similar revolution in software understanding tools, the current approach of manual RE, even with an expanding pool of SMEs, will continue to fall farther behind.

Although alternatives to software understanding do provide benefit, they are not nearly sufficient to address mission critical concerns in NS&CI missions.

6.3 Pursue Automated, Rigorous Technical Solutions to Software Understanding

Argument in Favor. The ultimate basis of whether software may imperil an NS&CI mission or not is to be found in its behavior. If that behavior will not threaten the mission, then it doesn't matter who wrote it, what development practices they used, whether it was digitally signed, whether attestation paperwork is available, etc. But if the software has behavior that could threaten the mission, mission owners have a critical need to know what that behavior is, even if the software was authored by trusted developers, leveraging the latest and greatest development practices, digitally signed, accompanied with certificates of attestation, etc.

There are several good reasons to think that a vastly reduced software understanding gap is possible. First, there is a potential coupling between the software production and understanding curves of Figure 2 that has yet to be adequately explored. Software understanding tools are, themselves, just software. The same advances that have produced the explosion in productivity in making software have yet to be fully harnessed for the purpose of creating an ecosystem of tools for software analysis. Second, mathematical techniques for modeling and reasoning about potential program behavior (often called "formal methods") have been advancing steadily in technical capability and scalability, both from academic innovations and from investment by companies like Microsoft, Amazon, and Google. The power of formal methods in software construction has reached the point where mathematical correctness proofs are possible for a subset of real mission questions and real mission software. Third, overall investment in software analysis capabilities has been many orders of magnitude lower than that in software development for many decades. Additional investment should be expected to bring significant advances. Finally, recent U.S. government actions aim to improve development practices; expanding on these by developing a new *Design for Analysis* program of guidance for developers of NS&CI system software could further reduce the software understanding gap by advocating to alter software design and development practices to facilitate analysis of the final software executable. With proper national investment, coordination, and vision, software understanding capabilities for NS&CI missions could be radically improved (see Section 7 for more details).

Only a technical analysis of the software itself correlates closely with the needs of NS&CI mission owners to have high confidence in the software upon which their missions and modern society and government depend. Given the momentous nature of NS&CI missions, missions owners need the ability to routinely ask any mission-related question of all mission-relevant software and receive rigorous, reliable, rapid, and repeatable answers. Of all the available options, only a rich, extensive, and growing ecosystem of components, libraries, and tools supporting software understanding would have a possibility of truly closing the software understanding gap.

Critique of the Argument. Of the three options, this one is the most challenging and costly. It has a high risk of consuming considerable resources for an uncertain benefit. The question of undecidability (see Section 4.3.4) raises the concern that resulting technical capabilities might be extremely impoverished. The investments required to create a growing ecosystem of reusable software analysis components, libraries, frameworks, and tools may not yield a dividend for some years, raising serious concerns about the expected return on investment (ROI). It's not clear that the risk to mission, society, and the government from the software understanding gap warrants the expense of this option.

6.4 A National Crossroads

The nation is at a crossroad with respect to its use of software. The need for software understanding and the reasons why the software understand gap exists have been illustrated throughout this report. Some of those reasons have roots in the characteristics of software, others in the chronology of software development in which analysis was never an equal partner to construction. The software understanding gap forms the key challenge at this crossroads, with the three options outlined above creating a choice. This choice will have national-level ramifications.

The first option, accepting the software understanding gap as inevitable, is based on a hope that future costs to society of software failures remain acceptable. The second option, redoubling efforts in software understanding alternatives, leverages ready-at-hand options, but can provide only modest reduction in the software understanding gap. Only the third option is poised to address the root cause giving rise to the software understanding gap.

For NS&CI missions, this problem will not be addressed naturally over time by the market. As the software production curve in Figure 2 has risen over the years, so has the software understanding curve, the result of steady progress by both academic and commercial communities. However, consumer systems do not require the level of rigor that society demands of NS&CI missions. The consumer market will never be willing to bear the costs of creating tools to address the software understanding gap to the levels required for NS&CI mission success, especially in the face of advanced, persistent threats. In contrast, national-scale investments to close the software understanding gap would have tremendous positive impact on the economy by reducing the multi-trillion-dollar fiscal burden of software failure and malicious cyber activity, including data breaches and ransomware (see Section 3.2.4). Much of the progress in software understanding over the years is likely already the result of federal research grants and other investments.

Perfection is not necessary to provide revolutionary value. To the extent that automated tools can help bring unexpected behavior to light prior to using software in an NS&CI mission, they will provide value. Thus, although we know that we cannot completely eliminate the software understand gap, incremental progress in highly consequential NS&CI systems is useful.

The question is to what extent the software understanding gap could be reduced in practice by a collective set of software analysis techniques, including the tools and techniques that are available today and those that could be created with additional investment in coming years. We cannot know this answer with certainty until we make the attempt. But, like the problem of cancer, some problems are important enough to make a sustained attempt, even if the rate of progress is unknown at the outset.

The questions raised about this third option are significant and are rooted in highly technical matters related to the nature of software as described in Section 4. Is an automated technical solution to software understanding truly possible? Is it feasible? What kind of ROI can we expect? To

consider questions like these, and others raised above in the critique of option 3, the authors of this report and five USG representatives brought together for the first time mission stakeholders and software analysis technical SMEs from across the government in a week-long workshop. The workshop focused on the following hypothesis about addressing the software understanding gap:

> *A revolutionary improvement to the U.S. government's software understanding capabilities is technically feasible for answering many questions against many systems to enable many missions; however, multiple challenges – technical, organization, funding, policy, and other – have prevented sufficient progress for years and will continue to do so absent intentional change.*

This hypothesis seeded the initial discussions of the workshop and set the stage for a deep discussion of whether a revolutionary capability is possible and, if so, what a national software understanding strategy would need to consider.

7 The SUNS 2023 Workshop and Results

The previous section described some of the thinking that drove the authors of this report, with funding provided by the Department of Homeland Security, to spend time in 2022 and early 2023 planning a workshop focused on software understanding for national security and critical infrastructure. The workshop took place in March 2023.

Of the options discussed in the previous section, that of pursuing rigorous, reliable, repeatable, and rapid software understanding through automated tools has the best chance of shrinking the software understanding gap. The challenges of software understanding are nearly universal across NS&CI agencies, their mission questions, and the software systems they need to analyze. Similarly, the technical solutions needed across each of these domains share significant commonality but are not currently being pursued in a way that can scale across agencies, mission questions, and software systems. A growing ecosystem of analysis components, libraries, and tools could be designed using an investment-style strategy, identifying the widely reusable components, creating standardized interfaces and data exchange formats, designing interchangeable parts with varying characteristics for a variety of uses, etc. If possible, such an ecosystem would fundamentally change the shape of the software understanding curve in Figure 2, accelerating analysis capability development and beginning to bridge the software understanding gap.

Based on decades of collective experience working on software understanding analysis tools for different NS&CI missions, the authors of this paper began to formulate a hypothesis that there is a set of common software analysis capabilities that could radically advance the nation's ability to reason about and understand software to answer a wide range of national security questions against a wide range of software. Through conversations with the government co-conveners listed below (see Section 7.3.1), the Software Understanding for National Security (SUNS) 2023 Workshop was conceived to consider and elicit the opinions of technical SMEs on these matters.

Appendix B provides answers to Frequently Asked Questions (FAQs) regarding the workshop's scope, terminology, and more. Workshop participants received these FAQs in advance of the workshop. Notably, FAQ 16 highlighted for participants that the purpose of the workshop was not to achieve consensus on the matters being discussed but to collect the SMEs' individual opinions.

The remainder of this section will go into detail about the workshop's key questions, the participating organizations, the structure, and the ultimate outcomes.

7.1 Editorial Note

The authors of this report are a group of SMEs from Sandia National Laboratories who proposed, organized, and executed the SUNS Workshop. The key findings in this section resulted from the authors' analysis of the opinions expressed by the SUNS 2023 SME participants in technical statements, during workshop discussions, and through informal "straw polls" that took place throughout the five days. The conclusions and recommendations in Section 8 are those of the authors. Specifically, Section 8 takes into account a variety of inputs and the authors' own expertise in addition to the key findings from the workshop.

The SUNS 2023 Workshop covered a vast scope of material in a short period of time. The technical statements written by the SMEs were particularly nuanced, filled with jargon, and presumed the context of the discussion. There was typically inadequate time to refine wording with precision or with a view toward a general audience. In this report, minor edits have been made to some of the workshop statements, poll questions, and recommendations in order to correct grammar, define acronyms, and provide clarity for a wider audience while making every effort to retain the original intent.

7.2 Key Questions

The SUNS 2023 Workshop was convened for the purpose of discussing this shared need in software understanding, the extent to which a scalable ecosystem approach might be technically feasible, and possible paths forward for tackling this national challenge. Participants also discussed the technical impediments, necessary stakeholder support, and near-term research and development (R&D) gaps.

Although deliberations on policy are essential to determining the nation's next steps in addressing the software understanding gap, the SUNS 2023 Workshop focused on soliciting the opinions of technical SMEs on the underlying technical landscape to inform such deliberations. As a result, the workshop prioritized closed technical discussions with subject matter experts during a three-day technical sprint with open sessions to begin and end the workshop. The discussions explored the challenges and possible paths forward for producing the technical capabilities required to meet NS&CI mission need for software understanding. There were three key questions to guide workshop discussions:

1. What are the technical impediments to an aspirational future state in which the nation has the capability to routinely analyze all relevant software, collecting evidence to execute missions with confidence?
2. What is needed to support a revolution in the USG's software analysis capability?
3. What are the key near-term R&D funding priorities for USG research organizations to support software understanding?

7.3 Participants

The SUNS 2023 Workshop involved a wide range of participants, including five U.S. government co-conveners, various mission stakeholders, and technical SMEs with experience in automated software analysis tool development. Over 90 individuals from 19 organizations attended the workshop, as described in more detail below.

7.3.1 Co-Conveners

The SUNS 2023 Workshop was convened by five government representatives, listed here alphabetically:

- Christopher Butera, Technical Director, Cybersecurity Division, Cybersecurity and Infrastructure Security Agency (CISA)
- Edward Jakes, Director, Nuclear Enterprise Assurance Division, National Nuclear Safety Administration (NNSA)
- Dr. Garfield Jones, Associate Chief of Strategic Technology, Cybersecurity and Infrastructure Security Agency (CISA)
- Dr. Robert Runser, Technical Director of Research, National Security Agency (NSA)
- Neal Ziring, Technical Director of Cybersecurity, National Security Agency (NSA)

7.3.2 Open Session Participants

The open sessions consisted of over 90 attendees, representing the following organizations, listed alphabetically:

- Carnegie Mellon University, Software Engineering Institute (SEI)
- Cybersecurity and Infrastructure Security Agency (CISA)
- Defense Advanced Research Projects Agency (DARPA)
- Defense Intelligence Agency (DIA)
- Department of Homeland Security, Science & Technology Directorate (DHS S&T)
- Department of the Army
- Georgia Tech Research Institute (GTRI)
- Institute for Defense Analyses, Center for Computing Sciences (IDA/CCS)
- Lawrence Livermore National Laboratory (LLNL)
- MIT Lincoln Laboratory (MIT-LL)
- National Institute of Standards and Technology (NIST)
- National Security Agency (NSA)
- Office of Naval Research (ONR)
- Office of the Director of National Intelligence (ODNI)
- Office of the National Cyber Director (ONCD)
- Office of the Secretary of Defense (OSD)
- Pacific Northwest National Laboratory (PNNL)
- Sandia National Laboratories (SNL)
- Zeichner Risk Analytics (ZRA)

7.3.3 Closed Session Participants

The 29 members of the closed technical session included experts in the field of software understanding from government research, Federally Funded Research Centers (FFRDCs), and University Affiliated Research Centers (UARCs). The requirement for being qualified as a technical subject matter expert (SME) for this workshop was 5+ years hands-on experience as a developer creating automated tools for semantic (as opposed to structural) program understanding.[127]

The technical SMEs at SUNS 2023 were drawn from the following organizations, listed alphabetically:[128]

- Carnegie Mellon University, Software Engineering Institute (SEI)

---

[127] For a discussion of semantic vs. structural analysis, see the Frequently Asked Question #13 on page 82.

[128] The SUNS 2023 Workshop organizers recognize that there may be other government research groups with significant experience in automated tools for software understanding but which we were unable to identify to invite to the workshop. This underscores the lack of community across the government in this domain. If you are interested in ongoing discussions with the SUNS community, please contact suns@sandia.gov.

- Cybersecurity and Infrastructure Security Agency (CISA)
- Defense Advanced Research Projects Agency (DARPA)
- Georgia Tech Research Institute (GTRI)
- Institute for Defense Analyses, Center for Computing Sciences (IDA/CCS)
- Lawrence Livermore National Laboratory (LLNL)
- MIT Lincoln Laboratory (MIT-LL)
- National Security Agency (NSA)
- Pacific Northwest National Laboratory (PNNL)
- Sandia National Laboratories (SNL)

7.3.4 Panelists

The open sessions included a panel that discussed the needs, risks, and challenges for a revolutionary approach to software understanding for third-party and legacy software in NS&CI missions. The following individuals participated on the panel, listed alphabetically:

- Carlton Brooks, Technical Director, Nuclear Command and Control Systems (NCCS) Cybersecurity, National Security Agency (NSA)
- Cherylene Caddy, Deputy Assistant National Cyber Director, Office of the National Cyber Director (ONCD)
- Dr. Ryan Craven, Program Officer, Cyber S&T, Office of Naval Research (ONR)
- Bob Lord, Senior Technical Advisor, Cybersecurity and Infrastructure Security Agency (CISA)

7.4 Workshop Structure

To facilitate the deep technical discussions envisioned while also including a broader set of mission stakeholders, the workshop was split into open sessions that took place on Monday and Friday and limited, more technically focused sessions which occurred Tuesday through Thursday.

The Monday and Friday sessions included a broad audience of mission stakeholders, program managers, and decision-makers. Monday's sessions included keynote presentations from the co-conveners present, presentations on case studies in software understanding, and a panel session. Collectively, Monday's sessions provided attendees an opportunity to broadly consider the software understanding needs across multiple mission areas and the nature of the technical challenges. In contrast, Friday's sessions focused on sharing some key results from the technical discussions.

The technical sprint on Tuesday, Wednesday, and Thursday focused the 29 technical SMEs on conversations around the current capabilities, gaps, and possible future of automated software analysis tools. Breakout session topics discussed during the technical sprint included: non-technical impediments, technical challenges common to today's software understanding tools, technical strategies to address multiple mission questions and programs under test, approaches to developing a robust software analysis ecosystem, and recommended near-term research and development priorities.

7.5 Key Outcomes

The core question that motivated the creation of the SUNS 2023 Workshop was whether a growing ecosystem of software analysis components, libraries, and tools could be created that would radically improve our nation's ability to understand software across a range of mission questions and systems of interest. The workshop participants' positions on this core question and other

important topics were explored using several poll questions, the highlights of which we will present here, with a more complete account discussed in Appendix C .

A vast majority (and in some cases, all) of the technical SMEs present expressed opinions consistent with the following key outcomes:

1. A 10x-100x+ improvement in software understanding capabilities is possible, but progress is currently prevented by lack of a centralized vision, funding that is at least 10x too low, inability to collaborate, and other non-technical issues.
2. A national software understanding vision will need to include foundational research in areas related to reasoning about software.
3. Due to the commonality across analysis applications, investment in reusable infrastructure components would likely yield a 5-10x or more return on investment in the long term (10+ years).
4. A Software Understanding Technical Advisory Board (SUTAB) of technical SMEs in software understanding should be established to assist the government in creating a national vision and guiding R&D funding recommendations.
5. There are serious technical challenges to understanding software's potential behavior after it has been created. Although some problems in the domain have been proven to be undecidable (see Section 4.3.4 for more information on undecidability), this will not prevent significant advancement in capability.

During the closed sessions, the technical SMEs identified issues to "run up the flagpole" for special consideration by the co-conveners and the federal government. These "flagpole issues" were those the SMEs each felt were the most important impediments to address to make progress on a national software understanding capability. The top five flagpole issues identified by the SMEs were:

1. The need for a national vision and plan
2. The nature and focus of funding
3. The need for community
4. The need for sharing and collaboration
5. The need for challenge problems, benchmarks, and competitions

The authors provide a high-level summary of the technical discussions and opinions relating to each of these flagpole issues in the sections following. For each issue, several straw polls were devised and conducted by the SMEs to gather opinions. See Appendix C for a list of the straw polls and their results.

Section 7.12 presents an annotated collection of near-term R&D priorities identified by the SMEs as aligning with the long-term national needs in software understanding but which could be pursued prior to the establishment of a national research agenda.

7.6 Issue #1: National Vision and Plan

The problem and solution spaces of software understanding cross agency and mission boundaries. Individual agencies focused on addressing their own specific mission needs using their individual resources are disincentivized from seeking more reusable, generalizable solutions that scale to the national need. Solutions require foundational research, applied mission research, and engineering activities to cultivate the needed tool ecosystem. This is beyond any individual agency's resources and responsibilities.

Additionally, the tools needed to address the software understanding problem are deeply complex, requiring highly skilled labor. The community of technical SMEs with experience building automated tools for software understanding is currently fragmented. There are small research groups spread across many federal agencies, FFRDCs, UARCs, industry, and academia, but there is presently no effort to coordinate and align their activities to address the national needs in software understanding.

For the reasons listed above, individual agencies working on their own are, and will remain, unable to make significant progress in addressing the software understanding gap that confronts the nation.

To make the progress needed, the authors of this report believe that a centralized, coordinated, and adequately resourced effort must be established to define a cross-agency, national-level vision, create and maintain a research roadmap, bring together the existing communities, and coordinate across agencies. This will require, at a minimum, the following actions:

- Establish a single, central, federal authority in charge of coordinating the creation and execution of a national vision to radically improve the nation's software understanding capabilities.
- Allocate resources to this federal authority such that it can adequately fund the necessary software understanding research, development, and engineering activities and recommend policies necessary for effective community collaboration and sharing.
- Create a Software Understanding Technical Advisory Board (SUTAB), comprised of technical SMEs with experience in software understanding tool development, responsible for defining and periodically updating a National Software Understanding Technical Roadmap and related research agendas, establishing research priorities, and advising the central, federal authority on research, development, and engineering investments.

The SUNS 2023 straw poll results related to the national vision and plan are in Appendix C.2.

7.7 Issue #2: Nature and Focus of Funding

The SUNS 2023 Workshop SMEs discussed the topic of existing funding vehicles to support the creation of automated tools for software understanding. Most SUNS SMEs expressed the opinion that the overall funding level across the government is between 10x and 100x too low to close the software understanding gap.

In addition to insufficient funding level, the duration and constraints on funding stifles the ability to make meaningful progress towards a national software understanding capability. Most funding available to the SUNS SMEs today is through mission-driven funding vehicles, which typically have an expected impact horizon of 1-3 years, articulate a set of deliverables due upon completion, and explicitly or implicitly come with sharing limitations and processes. These pressures and short timelines result in premature attempts to achieve mission impact that causes necessary foundational activities to be cut short and undermines the community's ability to investigate the types of long-term solutions that could radically improve the nation's overall software understanding capabilities. Separate funding vehicles from different agencies make collaboration difficult enough that individual SUNS SMEs said that they almost never pursue it. Additionally, the solutions that need to be explored span research and non-research areas, with a significant amount of engineering required to support research infrastructure and experiments, requiring close cooperation between research and capability development funding.

Misalignment of calls for proposals and funding vehicles to the nature of the technical challenge is a major barrier to progress. It is the view of the authors of this report that successfully achieving the potential benefits of software understanding capabilities requires, at a minimum, the following actions:

- Increase overall funding for an ecosystem of automated tools for software understanding by a factor of ten.
- Extend the duration of software understanding funding to support longer-term efforts that involve multiple necessary steps before demonstrating results.[129]
- Create a Software Understanding Technical Advisory Board (SUTAB), comprised of technical SMEs with experience in software understanding tool development, to establish funding priorities and advise the government on investment decisions.
- Fund basic research in the foundational gaps in automated software modeling and reasoning needed to build software understanding capabilities.
- Dedicate funding for the development, engineering, and maintenance of reusable software understanding frameworks, to alleviate experimental setup burdens from software understanding research activities.
- Fund not only research, development, and engineering of tools, but also the regular, periodic meeting of the SUNS SME community, the collaborative infrastructure the community needs, and the development and publication of interfaces, standards, metrics, benchmarks, and data sets.

For more details on these points, see the funding-related polls in Appendix C.1 and Appendix C.3.

7.8 Issue #3: Community

Regardless of whether it may be technically possible to radically improve software understanding capabilities, the technical community is currently so fragmented into organizationally siloed groups that it is only able to make marginal progress. The SUNS 2023 Workshop was in fact the first time many of these like-minded researchers had met together as a group. Despite the current fragmentation across the community, there is strong desire across the SMEs to see a SUNS community established.

Successfully establishing a strong technical community in software understanding requires, at a minimum, the following actions:

- Establish a regularly scheduled set of technical exchange meetings for the community of technical SMEs working in software understanding.
- Establish regular interchanges between technical SMEs and mission stakeholders to discuss how advancing software understanding capabilities can enable mission success.
- Ensure that FFRDC and UARC technical SMEs have the sponsor support, contract language, and funding required to engage in the community.

For more details on this issue, see Appendix C.4.

[129] Although no explicit poll was taken on what the preferred funding duration would be, informal SME discussions mentioned durations of 5, 7, and 10 years.

7.9 Issue #4: Sharing and Collaboration

Although establishing a community of SUNS SMEs is useful, it alone is not enough. The problem of software understanding spans agencies and missions and is too large for any one group to tackle alone. The solutions will likewise span agencies and missions. Creating those solutions will require a collective effort across a variety of organizations, including government research organizations, FFRDCs, UARCs, academia, and industry.

To develop the technical capabilities needed by NS&CI missions, technical SMEs must have the ability to share tools and data sets with minimal friction and must be able to collaborate daily on common computing infrastructure to run experiments, develop and debug software tools, and compare results. Furthermore, they must be able to share with minimal friction a wide range of artifacts (e.g., software under development, documentation, program samples, benchmarks, test suites, experimental results).

Unfortunately, a large gap exists between the intentions of government leadership to share and the practical ability of technical SMEs to do so. Bureaucratic processes experienced today by the SUNS SMEs result in multi-month delays to receive approval to share tools or libraries, even if they have no mission-sensitive information. Anecdotally, based on SUNS SME experiences, even when policy and decisionmakers support information sharing within the government, current processes can be so burdensome that many SMEs no longer attempt it. Given the extremely limited pool of software understanding SMEs in government, the USG has a vested interest in keeping those SMEs focused on overcoming technical challenges rather than navigating bureaucracy or becoming so discouraged by the processes that they leave government. Given the unbounded risk the USG is currently facing in NS&CI missions because of the software understanding gap, this situation ought to be unacceptable to mission owners and policy changes should be sought immediately to enable such SMEs within the government to share non-mission-specific information with zero friction.

In addition to lower friction sharing policies, SMEs from different organizations also require the infrastructure to collaborate. For technical information that is neither Controlled Unclassified Information (CUI) nor Official Use Only (OUO), adequate technical solutions exist, but inconsistent approvals and use policies across the SUNS SMEs community bar their use for effective community collaboration. For many of the SUNS SMEs, sharing approval is currently required on a transactional basis: approval authorities regularly review each new release of information before it can be placed on servers beyond government control. When effective collaboration calls for information to be pushed to the collaborative infrastructure daily or even hourly, such fine-grained, transactional approval is fatal to the collaboration necessary to develop solutions that can close the software understanding gap.

For CUI and OUO information, adequate technical solutions exist but are not currently available to the SME community as no group is tasked or funded to provide community infrastructure. For government-provided infrastructure to be viable, the community must have confidence that the infrastructure will be technically adequate, accessible remotely in ways that don't stifle the technical work and come with sufficient maintenance and support. Furthermore, the technical SMEs must have confidence that collaborative infrastructure will be available in the long-term; they will be unlikely to invest the time to properly leverage the infrastructure if there are concerns that it will be deactivated before their research and develop activities are completed.

Successfully establishing the effective sharing policies and collaboration infrastructure requires, at a minimum, the following actions:

- Establish enduring, pre-approved sharing agreements that maximize sharing and collaboration:
    - Enable technical SMEs to share non-mission-sensitive artifacts with near zero per-transactional administrative burden. Enabling the sharing of research work product among trusted partners must be the default, not the exception.
    - Recognize tiers of sensitivity, with criteria for determining the tier for any given artifact and successively less sharing information burden at lower tiers of sensitivity. The bottom tier should empower SMEs to immediately share with zero per-transactional procedural burden.
    - Define tiers of trust for various partners, including government, FFRDC and UARC researchers, and academic and industrial partners; agreements must define what tiers of artifact sensitivity can be shared with which partners, maximizing opportunities to share with zero per-transactional procedural burden.
    - Establish default criteria for releasing artifacts as open source.
- Establish collaboration infrastructure that:
    - Enables the full range of interaction modes typical in computer science research and development activities, including sharing of software code, files and data sets, tools and libraries, interactive chat, knowledge repositories such as wikis or forums, and compute infrastructure for running experiments.
    - Is readily accessible to the technical SME community from their home organizations.
    - Is fully supported and maintained for the long term, able to serve as an enduring knowledge hub and archive of past development artifacts, experiments, and results.

For more details on these issues, see the SUNS 2023 Workshop straw polls results in Appendix C.5.

7.10 Issue #5: Challenge Problems, Benchmarks, and Competitions

This flagpole issue represents the sole *technical* R&D item (as opposed to non-technical) identified in the SUNS 2023 Workshop as a top five issue. Currently, mission owners and SMEs alike lack effective methods for comparing and measuring results of different software understanding tools. Tools today often target different aspects of different mission questions on different software samples, making comparisons of different techniques, approaches, and research ideas effectively impossible. This inability has the effect of obscuring the depth of the software understanding gap from mission owners. In other problem spaces, public challenge problems, benchmarks, and competitions have spurred community focus and progress.[130,131,132,133] Investment in common software understanding benchmarks, competitions, and challenge problems is needed to alleviate this problem and spur effective research communication. For more details on this issue, see the SUNS 2023 Workshop straw poll results in Appendix C.6.

This same issue can be found on the list of Near-Term R&D Priorities (Section 7.12, item #6).

---

[130] "Competition on Software Verification (SV-COMP)," https://sv-comp.sosy-lab.org/.
[131] "The International Satisfiability Modulo Theories Competition (SMT-COMP)," https://smt-comp.github.io/2023/.
[132] http://predictioncenter.org contains results for CASP1 (1994) to CASP15 (2022). The Journal "PROTEINS: Structure, Function, and Bioinformatics" contains numerous articles discussing CASP results and progress in the field.
[133] https://grand-challenge.org contains 330 challenges and 3733 algorithms as of November 2023. See https://grand-challenge.org/about/ for a history of the platform.

7.11 Skilled and Knowledgeable Personnel

A final issue worthy of note was raised in many of the discussions. The skills needed to build effective software understanding tools, to innovate the needed solutions, and to architect those solutions to maximize reuse across missions are rare. The SUNS 2023 Workshop had 29 technical SMEs in participation, which likely represent a significant fraction of the government, FFRDC, and UARC personnel with those skills. No concrete data is available, but the authors estimate this number to be a quarter to a third of the available population. The skills needed to design algorithms and build tools to analyze software are very specialized, leveraging the fields of both computer science and math, and require years to develop. The total pool of such personnel is currently quite small.

Successfully reducing the software understanding gap for NS&CI missions requires, at a minimum, the following actions:

- Expand the availability of curricula in academic institutions related to software understanding techniques through academic government programs and funding.
- Resolve sharing and collaboration issues (Flagpole Issue #4) so that students and employees at other institutions may be able to participate in discussion boards, become familiar with tools, leverage learning opportunities, and begin to contribute. Students who are able to become familiar with SUNS community tools will be able to "hit the ground running" when hired into an organization seeking to close the software understanding gap.
- Establish benchmarks, metrics, and competitions (Flagpole Issue #5) to represent USG software understanding interests to the academic and commercial communities in a way that will spur learning, research, and competition.

7.12 Near-Term R&D Priorities

Although significant work is still needed to create a national-level R&D roadmap in support of a national unified vision and plan, participants identified 14 near-term research priorities that could lay the foundations or otherwise advance software understanding in the meantime. These R&D priorities describe foundational research and analysis infrastructure research that would improve software understanding capabilities for a wide range of USG missions, as well as some specific mission application research ideas.

7.12.1 Foundational Research

#1: Rapid Environment Modeling

Challenge: Software understanding tools are often unable to demonstrate effectiveness and capability outside of the small set of examples that motivated their development. Software executes in an environment that impacts behavior and thus analysis tools frequently struggle to produce meaningful results when the environment is unaccounted for and handled via generalized abstractions. For example, when an external function is invoked, might it exit, modify memory, compute return values based on its inputs, etc.?

Typically constructing a model for a single external function is not particularly difficult—rather, the difficulty lies in the enormous number of models that must be constructed for the full set of mission software. For example, there are thousands of functions in the Windows API and hundreds of string functions in Unix. When an analysis tool is created, the focus is on capability, not broad applicability. As such, time and time again, models are lacking. Unfortunately, the nature of a model can be impacted by the nature of the analysis, limiting its applicability; and, although there is some benefit

in building a large collection of (potentially soon outdated) models, flexibility is also needed so that the models can be adapted and tailored to the analysis being performed.

R&D Priority Description: Perform research to enable rapid automated environment modeling. Provide a technical solution for rapid creation and customization of models for various aspects of the execution environment, to scale our analyses to work on more broad classes of software. Environments of interest include initial states of common execution environments, common operating system libraries and system calls, and hardware devices.

#2: Multi-Architecture Intermediate Representation for Software Understanding

Challenge: Tools are often constructed to be independent of specific instruction set architectures or programming languages—instead they perform analysis on code expressed using an intermediate representation (IR) derived from the actual code of interest. IRs are developed for many purposes (e.g., compilation, disassembly, decompilation) and many goals (e.g., ease of generation and manipulation, freedom of expression, integration with other tools, level of abstraction). Today, software understanding tools often use IRs created for other purposes and to support goals that are tangential to needs of software understanding.

R&D Priority Description: Design and develop an IR for multi-architecture static analysis that serves as a common ground for different analyses. The resulting IR must be designed for performing forensic behavioral or semantic analysis of software binaries from multiple architectures relevant to USG missions. The IR must be designed to be readily leveraged by a variety of analysis tools and designed to answer different mission questions. Supporting tools are also needed to translate binaries from a variety of architectures to this common IR.

#3: Understanding Tool Errors and Confidence

Challenge: When reverse engineers use tools to understand software, they need insight into what the tools are doing and discovering; not just "answers;" the problem is challenging, and tools make tradeoffs (e.g., in completeness and soundness) and often lack the ability to easily convey to their users what they have and have not accomplished.

R&D Priority Description: Research novel techniques for assessing analysis tools' relative confidence and error rates in results to boost human-tool usability. Develop novel analysis strategies with sample components that have the capability to characterize result confidence, result error rate, evidence for results, and provenance of results. Resulting strategies should be able to integrate into existing software understanding frameworks under development or in use by government agencies.

#4: Machine Learning Assisted Binary Understanding

Challenge: When reverse engineers use tools to understand binaries, the complexity of the problem means they value insight that is predictive in nature; they are willing to accept that the insight might be inaccurate because it is not based on sound analysis but rather on learning obtained from large data sets.

R&D Priority Description: Research the feasibility and efficacy of using machine learning (ML) and large language models (LLMs) to accelerate understanding the functionality of a binary by automatically annotating a binary disassembly and decompilation with semantic information about its likely functionality. Produce a prototype that can take in a binary executable and annotate all its functions with a brief human-readable description of the function's behavior. Experimentally evaluate the human-readable descriptions for accuracy.

7.12.2 Analysis Infrastructure

#5: Reachability Analysis Framework

Challenge: Software understanding tools often struggle to demonstrate effectiveness and capability outside of the small set of examples that motivated their development. One culprit is the presence of code that ultimately has no bearing on the tool's analysis (e.g., if the problematic code was not present, the tool would work). Alas the tool, not knowing this, gets stuck attempting to work out details that may require high degrees of precision in some portions of code but not throughout.

R&D Priority Description: Develop an extensible analysis framework designed to maximize reachability of custom, higher-level analyses. Since the utility of many mission-specific analyses grows with the number of program points reached, the framework should be modular such that different mission questions can be evaluated at the program points reached. The analysis must be designed to support a wide range of Programs Under Test (PUTs) and include customizable and configurable approaches to tune the granularity/precision of handling dynamic memory, indirect and exceptional control flow, inter-procedural control flow, and environment models.

#6: Tool Evaluation Benchmarks

Challenge: Many tool development communities benefit from sets of benchmarks that allow tool comparisons, educate practitioners, and motivate participation by academia and industry. For example, "CASP" in protein folding[132] and "Grand Challenge" for medical imaging.[133] Although academic benchmarks and competitions[130,131] have driven progress in some important problems of software analysis, for software understanding more broadly, no sufficient benchmarks exist.

R&D Priority Description: Produce a benchmark dataset of analyzable software to enable evaluation of the efficacy of tools for software understanding.

The benchmark dataset must include:

- Mission relevant software as well as much smaller software.
- Both source code and binary versions, whenever possible.
- Ground truth answers to software analysis results that tools can be evaluated against, such as function partitioning, correct disassembly, and overall program behavioral capabilities.

Evaluation must include both static and dynamic analysis tools.

#7: Core Infrastructure Interoperability

Challenge: While there are some standard concepts in the field of software understanding (e.g., control flow graphs, abstract syntax trees) few tools support any degree of interoperability. Today it is not possible to construct better tools by using "best of breed" approaches.

R&D Priority Description: Perform research to enable the interoperability of core analysis infrastructure tools. Design interfaces and standards to allow intermediate results of different tools to be feasibly exchanged, compared, and fused with data from other tools. Produce modifications to Ghidra, Rose, and at least one other open-source analysis framework to enable the import and export among tools of their respective function partitioning, disassembly, decompilation, defined data types, data structures, data boundaries, and resolved indirect control transfers.

#8: General Abstract Interpretation Framework

Challenge: Existing abstract interpretation frameworks are typically tied to specific missions and cannot easily be repurposed for other missions. Additionally, current open-source abstract interpretation frameworks are inadequate for NS&CI mission questions.

R&D Priority Description: Develop or extend an abstract interpretation framework to enable custom static analyses for a wide range of NS&CI mission questions. Existing abstract interpretation frameworks are typically tied to specific missions and need potentially extensive rework to decouple mission-specific aspects from the underlying general framework. The research must design and evaluate an abstract interpretation framework that can modularly support analyses for different mission questions, permitting domains for different mission questions or analysis features to be turned on and off depending on the analysis goals.

#9: Higher Level Language Decompilation

Challenge: Reverse engineering practitioners rely heavily on decompilation to aid in the understanding of machine code. State of the art decompilers are integrated into frameworks such as Ghidra and Ida Pro but these tools lack customization features as well as the ability to enhance their results by strengthening the underlying analysis they perform. What is needed is more modularity, and more clearly articulated strategies to facilitate iterative improvement.

R&D Priority Description: Develop tools to decompile binary artifacts into higher-level source languages (such as C++, Rust, Swift, Go) to accelerate human reverse engineering. Produce a decompiler that takes an executable binary produced from a high-level language compiler and recovers a significant fraction of the original language features. Evaluate the decompiler by comparing it to the ground truth of actual source code.

#10: Tools and Techniques Survey

Challenge: Software analysis today is very niche and learning opportunities are often highly compartmentalized. Rather than bring the best hammer to each nail, individuals often become very familiar with specific tools and techniques and see the field as a whole through a narrow lens of experience; breadth is hard to come by, especially given the demand and need for individuals with expertise.

R&D Priority Description: Generate a catalog of open-source, unclassified software analysis tools so they can be more easily understood and leveraged across organizations. For each tool, identify the mission question it seeks to answer, the analysis techniques it uses, the types of programs it can analyze, how mature it is, the limitations it has, the architectures it supports, and how to get started using it. Determine and document an assessment of which tools and techniques are best suited for different kinds of mission problems and systems.

#11: Type Recovery

Challenge: The process of compilation discards type information once data is placed in memory or registers and appropriately operated upon. Recovering variable and structure information is an important step that is necessary to gaining many types of understanding. Although many tools provide some support for recovering type and structure, the problem is challenging and made more so in the presence of modern compiler optimizations. There are many existing techniques, including both static and dynamic analysis, but, in general, modularity is low, the field is extremely niche, and existing solutions are lacking in integration and ease of use.

R&D Priority Description: Develop a tool that recovers type information from x86 binaries to enable higher-level analyses. Recovered type information must include primitive data types (such as int, float, char), structured data types, and allow for tracking of analyst-driven custom data types (such as labels for attacker-controlled or sensitive data). The tool must be interoperable with Ghidra, ingesting type information from Ghidra and exporting additionally recovered types to Ghidra.

7.12.3 Mission Application

#12: Mobile Application Information Leaks

Challenge: Modern mobile devices contain many sensors, including cameras, microphones, radio-frequency receivers for wireless connectivity, global positioning capabilities, and more. In order to ensure that operational details and vital mission information are not being leaked through compromised mobile applications, mission owners need to know under what conditions mobile applications may access such sensors.

R&D Priority Description: Develop a tool that tracks dataflow for mobile applications to find potential sensitive information leaks. The tool must discover possible data leaks from suspected sensitive data sources, such as the phone's location, to unwanted data sinks, such as being sent over the internet. The tool must be able to run across a collection of Android apps in APK format and identify potential sensitive information leaks.

#13: SBOM Verification for Industrial Control Systems (ICS)

Challenge: Government agencies need to verify that ICS and other systems running our nation's critical infrastructure do not contain additional malicious behaviors beyond their intended behaviors. One effective step in this direction would be verification of SBOMs.

R&D Priority Description: Develop techniques for verifying SBOMs in their attached software. Produce a tool that verifies that the software behaviors, capabilities, elements contained in the SBOM are also contained in the software, and that the software does not contain additional behaviors, capabilities, or elements not contained in the SBOM. Tailor this tool to work on ICS and evaluate this tool on an ICS dataset. The ICS dataset should be documented and released open source (with an OUO addendum as necessary) to enable other research activities.

#14: Automated Backdoor Detection

Challenge: There have been multiple instances in which critical software used by the USG has been discovered to have a backdoor. One of the most notable of these was the SolarWinds incident. Automated techniques are needed to rapidly analyze software to detect backdoors prior to putting the software into service.

R&D Priority Description: Research techniques to determine whether binary software has an authentication bypass, such as a backdoor. Produce an analysis tool that behaviorally evaluates commercial .NET and Java bytecode binaries to search for such behavior and present any supporting evidence.

8 Conclusions and Recommendations

As discussed in Section 2, software already pervades today's NS&CI missions, with new forms of software like AI and ML being rapidly embraced. This software has been repeatedly demonstrated to have unexpected behavior that can cause systems to crash, contain exploitable vulnerabilities or malicious functionality, or fail in other ways that can imperil the missions that depend on it. Such

unexpected behavior is all too common, primarily because society's collective ability to create software vastly outstrips our ability to analyze and understand it, leading to the software understanding gap.

As discussed in Sections 3, 4, and 5, software is unlike any other human construct. The software understanding gap arises out of the fundamental nature of software and its characteristics as well as the history of software. While software in the past was simple enough that humans could adequately reason about it, innovation over recent decades has fueled multiple dimensions of exponential growth, expanding the software understand gap at an accelerating rate. For many years, experts have warned of the challenges of inscrutable software. Modern software is already well beyond our collective ability to adequately understand, and the situation continues to deteriorate.

As discussed in Section 6, there are a variety of options available to the nation for responding to the growing software understanding gap. Many recent national-level policy decisions, directives, and strategies have focused on certain facets of the problem and taken action to adopt various alternatives. Although these efforts are commendable, currently missing from the national dialog is a full recognition of the root cause of the software understanding gap and a call for the novel technical capabilities necessary to address it. The nation lacks—and is not currently pursuing—the technical capabilities needed to adequately analyze the potential behavior of software to answer all relevant NS&CI mission questions about all mission critical software prior to placing that software into service.

Given the above observations and the critical nature of NS&CI missions to society and government, it is imperative that we have high confidence in those missions. Mission owners need to be able to routinely, rapidly, reliably, and rigorously ask and answer all mission-relevant questions about the behavior of all software upon which their missions depend. Answers to such questions must not rest on assumptions of trust or attestation approaches, but instead on the strength of technical capabilities, with risk decisions informed by technical evidence packages of the software's potential behavior, derived from forensic analysis of the software artifact itself.

A discussed in Section 7, the 2023 SUNS Workshop gathered software understanding SMEs from across government to consider these issues and the extent to which technical progress might be possible. Significantly, a vast majority of those SMEs expressed the opinion that a dramatic improvement in national software understanding capabilities is technically possible. However, potential progress is further prevented by lack of a centralized national vision, insufficient resources, and a host of non-technical impediments.

The challenge of understanding software spans numerous government agencies and sectors of society involved in NS&CI missions. Although there is significant overlap in the technical solutions needed across these agencies, currently, there is no effective coordination to pursue common research solutions. Barring concerted action at the national level, the technical solutions needed are unlikely to organically emerge, given the nature of the challenge.

Below are three recommendations from the authors of this report for addressing the national challenge to NS&CI missions from the use of inscrutable software.

8.1 Recommendation #1: Make a National Decision to Address the Software Understanding Gap

The U.S. government should pursue the technical capabilities needed to dramatically reduce the software understanding gap. Until it makes this decision,

- Existing pockets of isolated initiatives will remain unable to address this strategic gap in a way that broadly meets cross-agency needs.
- Existing policies and incentives will continue to prevent existing researchers from effectively collaborating to make substantial progress.
- Existing risk management frameworks will remain unable to adequately measure the inherent risk to mission from the potential behavior of modern software.

To address the software understanding gap, the U.S. government should designate software understanding as a national priority.

This recommendation draws from the extensive technical expertise of the SUNS 2023 Workshop SMEs, many of whom expressed the opinion that radically improved software understanding capabilities are possible and will have a substantial return on investment. This recommendation is made with an appreciation that a whole-of-government effort in software understanding is extremely difficult, but with the recognition that it is also necessary.

8.2 Recommendation #2: Establish a Cross-Agency Software Understanding for National Security Executive Council (SUNSEC)

The software understanding community in the government today is fragmented. There are small research groups spread across many federal agencies, FFRDCs, UARCs, industry, and universities that do this work, but there is no unifying vision for how this research should be organized, conducted, converged, and distributed to mission stakeholders. Individual agencies currently lack both the mission scope and the necessary resources to address the full software understanding challenge.

Because of the whole-of-government nature of the problem and the need for pursuing solutions that maximize reusability across diverse missions, rather than tasking an existing agency, the USG should establish a cross-agency Software Understanding for National Security Executive Council (SUNSEC). The USG should charter SUNSEC to oversee, coordinate, and drive the creation of the needed software understanding capabilities. The SUNSEC would have the responsibilities, authorities, and resources to accomplish the following:

1. Develop and maintain a national vision and technical roadmap for software understanding suitable to the extreme heterogeneity of software and mission questions relevant to NS&CI missions.[134]
2. Prioritize and direct resources for the research, development, and engineering of the needed software understanding capabilities, ensuring that incremental activities are aligned with strategic national needs.
3. Establish working groups, coordinated across the government, to address key issues, including metrics, benchmarks, standards, and common frameworks, utilizing both competitive and cooperative structures, as appropriate, and providing incentives for both public and private landmark advancements.
4. Establish a talent pipeline through academic institutions to expand the pool of required expertise.

[134] The authors of this report have produced an initial version of such a national-level technical roadmap: D. Ghormley, T. Amon, C. Harrison, and T. Loffredo, "Software Understanding for National Security (SUNS) Technical Research, Development, and Engineering Roadmap," Dec 10, 2024. Available at https://suns.sandia.gov.

5. Coordinate with mission stakeholders and research groups across government, industry, and academia, to ensure that research is progressing towards mission needs.
6. Engage with existing government agencies and communities of interest to identify policy and other non-technical barriers requiring adjustment to maximize cooperation, sharing, and collaboration across the software understanding community.
7. Establish and cultivate a thriving community of SMEs in software understanding across government, industry, and academia, defining tiers of trust for varying sensitivities.
8. Establish and support the technical infrastructure and central repositories necessary to minimize friction in sharing and collaborating across the software understanding community at the pace of technical exploration and development.
9. Advise the government on new cybersecurity, supply chain, open-source, and acquisition policy as software understanding capabilities advance.

The challenges involved in building automated software understanding tools are highly technical and specialized. Like the national efforts in cancer research that are being led by researchers and practitioners in oncology, software understanding research must be guided primarily by researchers and practitioners with deep experience in software understanding techniques and NS&CI mission needs in software analysis. The SUTAB recommended by many of the SUNS 2023 Workshop SMEs in Section 7 should be an integral aspect of the SUNSEC, guiding technical decisions in pursuing the national software understanding program and associated investments.

8.3 Recommendation #3: Direct the Coordination of Software Understanding Collaboration Policies and Activities Across Agencies

Although the U.S. government has existing SMEs with expertise in the types of techniques needed to reduce the software understanding gap, as described in Sections 7.6-7.10, that community is currently prevented from sharing and collaborating by inconsistent policies across institutions, incentives that are misaligned to the nature of the software understanding problem, and other non-technical barriers. In conjunction with the establishment of the SUNSEC, existing agencies will need to align policy, existing roadmaps, and ongoing investments to maximize support for the national vision in software understanding while protecting mission sensitivities.

8.4 Conclusions

The accelerating software understanding gap is driving the nation toward extensive, unmeasurable risk from our use of inscrutable software. Without high-level action, these risks will continue to permeate and weaken NS&CI missions, the government, and society. However, these risks are not inevitable. A vast majority of the 2023 SUNS Workshop SMEs expressed the opinion that radically improved software understanding capabilities are technically possible—if the nation decides to act. The U.S. government has extensive expertise available to support a national software understanding effort, but that expertise currently lacks cohesive direction, adequate resources, and is stifled by non-technical barriers and policies that prevent the formulation of an effective community.

As stated in the "Closing the Software Understanding Gap" report:

> *Closing the software understanding gap requires decisive and coordinated action across the U.S. Government. The challenge must be addressed through policy action, mission engagement, and technical innovation. Like previous national efforts to meet enormous challenges—the Manhattan project, the space race, and the war on cancer—addressing the software understanding gap requires effective organizational*

> *structures, subject matter expert-guided research direction, and an enduring and tenacious focus to address this urgent need.*
>
> *Enduring national attention is necessary to enable a whole-of-government approach to national security and critical infrastructure missions to address associated policy and technical and resource challenges.*[2]

Decisive action is needed or the software understanding gap will continue to threaten NS&CI mission success. A radically improved future for software understanding and technically informed mission risk is possible if we have the national will to make the decision to undertake the journey.

Appendix A Sample Mission Questions

A mission question is a high-level but specific technical question that a mission owner may pose about the software upon which their mission relies. These are top-down questions about the software and its behavior, driven by the mission concerns.

Answering some mission questions requires only identifying *anything of relevance* in the software under test, such as a malware mission that can reject a software sample if anything malicious is found, or a United States Cyber Command mission in which only one exploitable vulnerability is required. Although some software behaviors of interest may be easy to detect while others may be challenging, with these types of mission questions, finding anything relevant is often sufficient. Other missions depend more on finding *everything of relevance* to the question at hand, such as a cyber assurance mission. In these cases, the level of confidence depends on the level of thoroughness of the analysis, with complete confidence only being achieved if the analysis of the software is complete. Here are some examples of mission questions to illustrate:

1. Could my industrial control system software permit unauthorized control of the critical function?
2. Is there remote administration access? Is there an authentication backdoor?
3. Does this software permit a remote user to disable this critical functionality of the system (e.g., a remote kill switch)?
4. Does the software "phone home?" Under what conditions? What systems might it connect to?
5. Could this software write any files? Which directories might it write these files to? Which files could this program read?
6. Is there behavior tied to specified dates or times?
7. Is there behavior tied to specific geographic locations? (Important corollary: Does my guidance software behave differently in different regions of the world?)
8. Could any local files be sent out over the network?
9. Could this software record the user's keystrokes?
10. Could this software take screen shots?
11. Under what conditions can the camera/microphone be turned on?
12. Is the integrity of my critical data preserved by the software?
13. Are there inputs that cause the decision logic to lock up?
14. What network protocols does this software handle?
15. Is there encrypted communication? Is there a hardcoded key? Are the certificates checked properly? What certification authorities are trusted?
16. Is there communication that is unencrypted?
17. What input sensors are used to determine the action of this actuator?

Some mission questions are asked and answered today, at least partially. Here is a list of a few mission questions that are asked today:

1. Is there an input that will crash this program? (Fuzzing tools apply here.)
2. Are there things in this software that I have seen before and discovered were bad? (Antivirus software applies here.)
3. Does the program do what I designed it to do in this specific environment with this input? (Acceptance testing applies here.)
4. Has the program been modified since it was hashed or signed? (Hashing and signature checks apply here.)

Appendix B Frequently Asked Questions (FAQs)

There are two FAQs that were distributed prior to the workshop to address the purpose and scope of the workshop and to define key terms and concepts. A number of those questions and answers are reproduced here.

The points below attempt to provide simple answers to what can often be very complicated questions, many of which cannot be fully answered succinctly. We strive to inform and use simple vocabulary. The answers you see reflect the opinions of the organizers of the SUNS Workshop.

1. **Question: What is "software understanding" and why is it important?**
   Answer: For the purposes of the SUNS Workshop, "software understanding" is taken to mean the practice of discovering the behavior of software by analyzing the software artifact itself, rather than relying on documentation, developer attestation, or development processes. It is important because so many vital national security and critical infrastructure (NS&CI) functions of society rely on software, yet the way software is built today makes it inscrutable—it is difficult today to know what types of behaviors could occur when the software is used. Unexpected behavior in the software, whether intentional or not, can cause the system, which depends on that software, to fail or to be vulnerable to a hostile cyber actor; this in turn may cause the government mission relying on that system to fail. By using software that we cannot adequately understand, we have accepted unbounded risk in nearly every facet of government and society.

2. **Question: What is the purpose of the Software Understanding for National Security (SUNS) Workshop?**
   Answer: Recent years have seen the integration of software into nearly every NS&CI mission. Much of this software is not written by the government, but by third parties, and is therefore largely opaque to the government missions that depend on it. SolarWinds[135] and other incidents have demonstrated that, in many cases, we lack adequate capabilities to understand the potential behavior of the software upon which we so heavily rely. The purpose of this Workshop is to bring together a group of USG researchers with strong experience in semantic software analysis to discuss whether revolutionary software understanding capabilities might be possible that could address the nation's software analysis needs. Workshop discussions will range over the need, advisability, and the facets of a potential national agenda and needed national software analysis community to support it.

3. **Question: What is the scope of this workshop?**
   Answer: The workshop focus is on potential software understanding solutions that could be developed over the next 10 years to answer a wide range of mission questions about third-party software. All types of software artifacts in widespread use in NS&CI missions are in scope, including binaries, byte codes, intermediate forms, source code, etc. All types of systems and software in widespread use in these areas are in scope, including desktop, server, mobile, embedded computing, internet of things, industrial control systems, security systems, building automation, manufacturing, energy production or distribution, communication, vehicles, aviation, and many more. All layers of software in these systems are in scope, whether at the application, operating system, firmware, or other layers. Furthermore, the scope of this

[135] "Joint Statement by the Federal Bureau of Investigation (FBI), the Cybersecurity and Infrastructure Security Agency (CISA), the Office of the Director of National Intelligence (ODNI), and the National Security Agency (NSA)." Cybersecurity and Infrastructure Security Agency (CISA), 5 Jan. 2021. Press release.

Workshop is not on one single mission question but rather on the broad range of questions that various government missions need to ask about software (see question #6).

This is an enormous space, but it is the reality of the challenge. This enormity speaks to the kind of approaches that must be taken to achieve an acceptable return on investment (ROI). In too many cases today, mission-driven analysis investments result in a monolithic tool designed to solve a specific problem using fixed techniques on a particular type of system. The next tool development activity must start from scratch, or nearly so. These monolithic approaches mean that answering the vast array of mission questions (see Question #8) on all the systems above (the "cross-product" approach) would cost more than the GDP of the nation. A different type of approach—which focuses on reusable components, instantiated/tuned/adapted to the particular mission question and system under analysis—may be able to deliver an ROI that makes sense.

Fortunately, it is not necessary to address *all* of the systems listed above in order to achieve an acceptable ROI; even being able to analyze 50% of them would be a revolutionary advance. Ideally, discussions would focus on identifying approaches to maximize the number of mission questions that can be answered and the number of systems for which we can answer them. The following areas, however, are out of scope:

- Machine learning algorithms as the software under study because there are already established communities studying them and their particular characteristics. They are in scope when leveraged as techniques to understand other software.
- Analysis of software running on non-conventional architectures (e.g., quantum computers, neuromorphic computers, etc.).
- Operational network artifacts (such as network packets, emails, and the like) and operational system artifacts (such as logs and configuration files) are out of scope as the principal objects of analysis. Analyzing software to understand under what conditions a particular network packet or log entry could have been generated could be in scope, though.
- Unexpected analog effects which violate the foundational assumptions of digital systems (such as row hammer).
- Side channels from software or processor execution.

4. **Question: Aren't some of these problems (particularly supply chain) amenable to policy changes that can help? Are those in scope for this workshop?**
   Answer: We agree that policy changes may be possible in a variety of areas that would enable us to improve the nation's ability to answer mission questions about software. This workshop is structured around technical conversations involving a particular type of expertise—these technical SMEs are not policy experts and therefore policy is not a central focus of this workshop. However, if the technical discussions identify significant potential capability advances that depend upon certain policy changes, we will capture those in the workshop report to inform and spur further conversations with policy makers.

5. **Question: Is this workshop about software assurance?**
   Answer: Not directly, but there may be interest in leveraging the results of this workshop for software assurance questions, subject to certain limitations. In its strongest form, software assurance seeks to provide strong guarantees that software has no undesirable behaviors, including vulnerabilities, that could put the mission using the software at risk. Unless a particular software package was designed to be readily analyzable, software understanding techniques are

extremely limited in their ability to prove that certain behaviors are absent; instead, they seek to identify what behavior may be present, but typically cannot guarantee that they have found all behaviors that are present.

However, mission owners routinely make decisions about software that should be highly assured, but for which that assurance is not practical or possible to achieve today. Currently, these decisions are, typically, not informed by a body of technical evidence derived from an analysis of the software itself about its possible behavior. In this environment, software understanding would be a material advancement over the approaches currently being taken but should be understood by mission owners as offering substantively weaker answers than high assurance approaches such as formal methods.

6. **Question**: **Can software understanding be useful if it cannot provide assurance?**
   Answer: First, note that there are both high and low consequence systems that use software for which we do not have assurance guarantees. Risks should be managed (even if they cannot be quantified), and mission owners routinely take action to reduce risk. Software understanding can be useful as part of these risk reduction measures. Software understanding is better suited towards missions that seek "evidence" for behavior that is present or that can accept a partial understanding of the system. Transforming an assurance question about the lack of behavior into one or more understanding question about the presence of behavior can be a useful assurance exercise. It is important to remember that when software understanding tools fail to find evidence for the presence of a behavior, they are by no means ruling it out.

7. **Question**: Is this workshop about finding software vulnerabilities?
   Answer: Software vulnerabilities are in scope, but the workshop is by no means limited to them. There are many mission questions listed in Question #8 that are not focused on finding vulnerabilities.

8. **Question: What do you mean by "mission questions"? Can you give some examples?**
   Answer: A mission question is a high-level but specific technical question that a mission owner may pose about the software upon which their mission relies. These are top-down questions driven by the mission concerns. Here are some examples:

   - Could my control system software permit unauthorized control of this critical function?
   - Is there remote administration access? Is there an authentication backdoor?
   - Does this software permit a remote user to disable this critical functionality of the system (e.g., a remote kill switch)?
   - Does the software "phone home?" Under what conditions? What systems might it connect to?
   - Could this software write any files? Which directories might it write these files to? Which files could this program read?
   - Is there behavior tied to specified dates or times?
   - Is there behavior tied to specific geographic locations? (Important corollary: Does my guidance software behave differently in different regions of the world?)
   - Could any local files be sent out over the network?
   - Could this software record the user's keystrokes?
   - Could this software take screen shots?
   - Under what conditions can the camera/microphone be turned on?

- Is the integrity of my critical data preserved by the software?
- Are there inputs that crash this program?
- What network protocols does this software handle?
- Is there encrypted communication? Is there a hardcoded key? Are the certificates checked properly? What certification authorities are trusted?
- Is there communication that is unencrypted?
- What input sensors are used to determine the action of this actuator?

A few mission questions are asked today, including these examples:

- Is there an input that will crash this program? (Fuzzing tools apply here.)
- Are there things in this software that I have seen before and discovered were bad? (Antivirus software applies here.)
- Does the program do what I expect it to do in this specific environment with this input? (Acceptance testing applied here.)
- Has the program been modified since it was signed? (Hashing and signature checks apply here.)

9. **Question: Are there other questions beyond mission questions that software understanding can help address?**
   Answer: Yes. Since few automated tools exist today that can help answer the high-level mission questions listed above, missions that have a compelling need to analyze software (and the resources to fund it) employ reverse engineers to manually analyze the software to seek evidence to inform the mission question. In principle, software understanding tools could provide low-level information to such analysts that would radically accelerate their manual efforts. Here are some examples of low-level questions about software:
    - What input values enable me to reach a specific program point?
    - Which code runs when this input is provided to the program?
    - What code must have run to generate this specific output?
    - Given a set of inputs and a distinguished input, which code would run in processing the distinguished input that didn't run for the others? (Same for outputs.)
    - Could this code throw an exception? If so, where is the exception handler?
    - What are the potential targets of this computed call/jump/return?
    - What are the potential addresses of this pointer?
    - What are possible values of this parameter? Register? Memory cell?
    - Given two points in the code, under what conditions can execution get from A to B? What inputs might affect those conditions?
    - Can program execution flow from point A to B without going through C?
    - Comparing a new version to an older version, what behaviors are new?

10. **Question: Where does the SBOM fit within Software Understanding?**
    Answer: A Software Bill of Materials (SBOM) is a list of the software components that make up a particular software application. While an SBOM can be useful in understanding what some of the functionality in a particular software application may be, there remain many questions about possible software behavior that the SBOM alone cannot answer. In effect, one particular mission question may be "Is this SBOM correct?" Another may be "What is the SBOM for this software?" For examples of mission questions that go beyond the SBOM, see the previous question, above.

11. **Question: Is this workshop about source code analysis or binary/byte-code analysis?**
    Answer: This workshop is about analyzing the systems described in Question #3. In certain cases, source code (or partial source code) may be available, but in a great many mission scenarios and systems, it is not. As the SolarWinds event demonstrated, even when source code is available, it may not reflect important behavior present in the binary. Consequently, the workshop will be primarily concerned with software artifacts as typically put in use (binaries, byte-codes, etc.), though it is of interest how capabilities can be improved in the cases where source code (or alleged source code) is also available.

12. **Question**: **Are efforts to build software more securely in the first place in scope for this workshop?**
    Answer: Designing better, more secure forward engineering processes and practices is an important and relevant topic, especially in cases where the government has control over the development of software they are using; for example, critical software in nuclear weapon systems. However, that is not intended to be the focus of this workshop. Numerous missions of the government need to answer questions about the potential behavior of third-party software for which we do not have direct control over development and must therefore rely on technical analyses. The work done in the space of verifying forward engineering—see Facebook's Infer for example—is very important, but this workshop is principally focused on analyzing third-party software.

13. **Question**: **What is "behavioral analysis" (or "semantic analysis") and why is it important?**
    Answer: One way of categorizing software analyses is by whether they are *structural* or *semantic* (also called *behavioral*). An analogy may be useful. In algebra, the following equations are all considered equivalent:

    $$x=y+4 \quad x=4+y \quad x-4=y \quad x-y=4 \quad x-2=y+2$$

    These all have different structures (different order of the symbols and sometimes even different symbols), yet they all have the same meaning (*x* is four greater than *y*). Conversely, you can have equations with similar structure but with different meaning, for example:

    $$x=y+4. \ y=x+4$$

    The same is true in software: different structures may have the same meaning, and similar structures can have very different meanings.

    In software analysis, *structural* analyses focus on the bytes or structures that make up the software, while *semantic* (or *behavioral*) analyses focus on what the software means in terms of what it does, its behavior (not how it is represented).

    In many cases, structural analyses can be much faster, cheaper, and scalable than semantic analyses, but they are more limited in what they can do. Since many important mission questions (see Question #8 above) require analyses that can analyze the behavior of the software, that is the focus of this workshop, though structural analyses may be able to provide useful information to assist behavioral analyses.

14. **Question**: **Is this Workshop meant to exclude dynamic analysis and focus only on static analysis?**
    Answer: Absolutely not! Although there is a lot of software understanding research leveraging static analysis, dynamic analysis is powerful, useful, and often complimentary to static analysis. Both dynamic and static analysis offer unique benefits in the software understanding problem

space. Talking about the role of static, dynamic, and hybrid analyses is the sort of thing that the technical SMEs will likely discuss during the workshop.

15. **Question**: **What about industry and academia? Shouldn't the workshop include them?**
Answer: They absolutely have a role to play in developing and improving the capabilities contemplated by this workshop. However, for an initial discussion, the agreement with the co-conveners has been that the discussions will be limited to researchers employed by the government and FFRDCs/UARCs. Depending on the recommendations coming out of this workshop, there may be future workshops that will likely include academia and industry.

16. **Question**: **What exactly are the government's needs with regards to software understanding? Do the Technical SMEs know?**
Answer: The technical SMEs selected for this workshop have, collectively, extensive experience with a variety of government missions to draw upon. Additionally, the government representatives who are co-convening this workshop will provide keynote talks to set the stage on the first day. The workshop organizers believe that many times government mission stakeholders limit their expectations with regards to software analysis to the current state of the practice, or incremental improvements of it. The purpose of this workshop is not to think evolutionarily, but revolutionarily about what future capabilities might be possible with sufficient national focus. This is not a question that government mission stakeholders are best positioned to answer.

17. **Question**: **The way software analysis tools are often built today is with specific funding, for specific customers, on specific problems. Is this workshop suggesting that a "general analysis for everyone" approach can work?**
Answer: Not exactly. Given the mission questions laid out in Question #8 and the scope described in Question #3, there can never be a single tool that can do all that is needed. However, with a different model about how software analysis capabilities are funded and pursued, something radically different may be possible. Very little software that is used today is written fully from scratch—it is often composed from various libraries using various development frameworks. Today there is no analogy of this process focused on composing capabilities for software understanding. It is likely that there is a common investment in underlying analysis components and frameworks that would pay dividends by enabling the rapid development of specific software analysis tools for specific mission questions and specific programs under test. The idea of reusable analysis components is a difficult and delicate balance that is hard to get right. Too much focusing on infrastructure, and you might lose sight of what problem you are solving; too much focusing on the problem, and you might not be able to build common infrastructure that would benefit multiple missions in answering a broad range of questions. Sorting through this thorny issue and deciding if it is even tractable is the sort of thing that will be discussed at the workshop.

18. **Question**: **What is "undecidability" and what does it have to do with software understanding?**
Answer: Informally, "undecidable" is a term used in computer science theory to refer to a special kind of impossible problem. The concept of "decidability" relates to whether or not it is possible to construct an algorithm that is guaranteed to always answer a particular type of question (specifically, a *decision problem*) about a program. A decision problem is a yes/no question about the behavior of a program over all possible inputs; if that question is always answerable for all

inputs in finite time, then it is *decidable*. If there is no algorithm that is guaranteed to produce such an answer, then the problem is *undecidable*. That is, the question of decidability is one of universal guaranteed success.

The most famous example of undecidability was given by Alan Turing in what is known as the Halting Problem,[136] a decision problem asking whether a program will halt regardless of its input. Turing proved that this question is undecidable. Rice's theorem,[137] a generalization of this result, states that all non-trivial semantic properties of programs are also undecidable. Both of these refer to this concept of decidability.

To call a decision problem undecidable only requires that a single program be identified which defies an algorithmic determination of the answer. In Turing's proof of the Halting Problem, he crafted a program which sometimes halts and sometimes does not depending on its input. This single instance proved that you cannot *always* determine whether a program halts (because it doesn't work on his specially crafted one).

Fortunately, NS&CI missions do not require a universal guarantee of success to benefit from analyzing the software used in their mission spaces. Saying that something *cannot always work* does not mean that it *will never work*. The question to be answered may be both undecidable *and* answerable in practice for a great many programs of interest. By instead focusing on instances of universal generalities, it is possible to identify many non-terminating conditions, the Halting Problem notwithstanding.[138] Asking mission-relevant questions about a specific software program can be tractable, as demonstrated by successes in the automotive aviation, and space industries.[139] Such examples demonstrate not only that instances can be decidable, but also that they can be tractable and cost effective.

[136] See https://brilliant.org/wiki/halting-problem/.
[137] See https://en.wikipedia.org/wiki/Rice%27s_theorem.
[138] For an example of a software analysis tool used to analyze safety of nuclear power plant software, see Alain Ourghanlian. Evaluation of Static Analysis Tools Used to Assess Software Important to Nuclear Power Plant Safety. Nuclear Engineering and Technology, 47(2):212-218, 2015.
[139] See https://www.astree.ens.fr/.

Appendix C Straw Poll Results

This section details the straw poll questions used to tease out SME opinions on a variety of issues related to the importance, challenges, and needs of a national effort to improve software understanding capabilities.

The SMEs themselves quickly drafted the poll questions listed below and considered them in a single afternoon session. It is important to acknowledge the process was informal, lacked consistency, was not scientifically rigorous, and was intended as a shorthand to reflect the opinions of the SMEs and not to achieve consensus among the SMEs.

C.1 Poll Results: The Incoming SUNS Hypothesis

This set of poll questions relates to the hypothesis that led to the SUNS 2023 workshop, discussed in Section 7. These questions were asked to ascertain workshop SME opinions on various aspects of it.

1. **What is holding us (the software understanding SME community) back more?**

   Technical impediments: 4 SMEs
   Non-technical impediments: 17 SMEs

   Editor's note: There are numerous technical challenges in creating a tool to analyze software to seek evidence related to some mission question. Prior to SUNS 2023, an open question was to what extent those technical challenges are currently the main inhibiting factor toward progress. The results of this poll indicate that >80% of the SMEs voting feel that non-technical impediments dominate concerns about making forward progress. This is good news from a technical standpoint, because this means that organizational change alone may enable the community to accelerate its rate of progress.

2. **If the non-technical impediments were fully removed, how much would you guess we could increase software understanding capability in 10 years?**

   1x-10: 5
   10-100x: 13
   100x-1000x: 4
   1000x+: 1

   Editor's note: Based on what is understood technically today, the SUNS technical SMEs feel that 1-2 orders of magnitude improvement is technically possible in creating automated tools for semantic program understanding. This gives us an initial estimate of how much progress might be expected if the non-technical impediments can be removed.

3. **What level of funding do you believe is needed for software understanding?**

   We are already investing too much here: 0
   Funding is fine: 1
   10x more funding is needed: 16
   100x more funding is needed: 9

   Editor's note: It is notable that over 1/3 of the technical SMEs believe that 100x more funding is required to fully realize the technical progress available.

4. **Given the more-than-two-trillion-dollar annual cost of poor software quality[140] and the predominance of legacy and 3rd-party software in so many segments of national security and counterintelligence, what net return on investment (ROI) do you believe government investment in improved software understanding could produce for the economy in 10 years?**

   Poor (ROI of <1.0): 1
   Good (ROI of 1.0-2.0): 7
   Very Good (ROI of 2.0-5.0): 13
   Excellent (ROI of 5.0+): 2
   Abstain: 1

   Editor's note: Nearly all technical SMEs believe that government investment in improved software understanding will more than pay for itself through reduced burden to the economy by enabling undesirable behaviors to be better identified prior to putting software into use.

5. **Will undecidability prevent significant advancement in capability?**

   Yes: 2
   No: 20
   Abstain: 3

   Editor's note: For more information on the term "undecidability", see Frequently Asked Question #18. The implication of this poll is that the SUNS SMEs feel that significant advancements in capability can be made before the particular computer science limitation of "undecidability" blocks progress.

6. **We may be in a software understanding arms race with our adversaries.**

   Agree: 22
   Disagree: 1
   Abstain: 3

   Editor's note: The way most software is written results in unintended behavior. This is typically true even of the software used in NS&CI systems. A disparity in the ability of two nations to analyze their own and other nation's software can lead to national advantage. If an adversary learns to analyze the software used in US NS&CI systems better than the US can analyze them itself, this can lead to strategic disadvantage for the USG. Other polls in this document indicate that the SUNS SMEs believe that current USG investments are inadequate for this arms race.

7. **If the only change was to enable our communities to collaborate as seamlessly as we want, how much improvement might be made to software understanding capabilities in the next 10 years?**

   1-5x: 8
   5-10x: 12
   10-50x: 4
   50-100x: 0
   100x+: 0

[140] Herb Krasner, New Research: The Cost Of Poor Software Quality in the US: A 2022 Report, The Consortium for Information & Software Quality (CISQ), https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/.

Editor's note: One of the impacts of the non-technical impediments recorded in this document is considerable difficulty in collaboration within the SUNS community. This poll was designed to assess that impact.

8. **______________software understanding capabilities might be possible, but progress is currently prevented by non-technical issues, funding that is 10x+ too low, and lack of a centralized vision.**

   Improved ~5x: 4
   Dramatically improved ~10x: 7
   Radically improved ~100x: 11
   Revolutionary: 1

   Editor's note: This poll was designed to determine what description the SUNS SMEs preferred to describe the potential for progress.

C.2 Poll Results: The Need for a Vision

This set of polls was designed to determine what the SUNS SMEs think is necessary to form and support a unified vision for software understanding across the whole of government.

1. **A Software Understanding Technical Advisory Board (SUTAB) of technical SMEs ____ should generate the national software understanding vision.**

   Alone: 0
   Plus stakeholders: 2
   Plus stakeholders plus practitioners: 14
   Plus practitioners: 7

   Editor's Note: This poll was designed to ask who should compose the membership of a proposed federal advisory board around software understanding. Most SMEs thought the board should be inclusive of mission stakeholders and mission practitioners, so that the technical agenda can be informed by closer ties to mission questions.

2. **The Software Understanding Technical Advisory Board will _____**

   Strictly prioritize topics: 0
   Make recommendations to whoever oversees the funds: 17
   Oversee funds: 2

   Editor's note: Unlike the national cancer board, the SUNS SMEs do not feel that the SUTAB alone should direct funding. Rather, the overwhelming opinion was that the SUTAB should work in collaboration with funding authorities and make recommendations on research investments.

3. **The scope of the national software understanding vision should include non-cybersecurity mission spaces.**

   Agree: 24
   Disagree: 0

   Editor's note: Software understanding capabilities can be used to gather evidence related to cyber security questions but can also be used more general to understand the potential behavior of systems in non-security contexts. This poll indicates the SUNS SMEs unanimous belief that software understanding capabilities apply to missions beyond cybersecurity, which likely is factored into the return-on-investment expectations of Poll #4 on page 91.

4. **The scope of the national software understanding vision should include foundational research.**

   Agree 24
   Disagree: 0

   Editor's note: This poll indicates the SUNS SMEs believe that foundational research is required to achieve the improvements and return-on-investment discussed in this document. While parts of the research foundation in software understanding is in place, there are numerous gaps which must be intentionally addressed.

5. **The research agenda derived from the national software understanding vision should be maintained into the future by revisiting it every ___.**

   One year: 5
   Five years: 17
   Ten years: 0
   Disagree: 1

   Editor's note: The majority of SUNS SMEs believe that a vision for software understanding research must be revisited and possibly revised at least once every several years.

6. **Software understanding missions will fail absent strong coupling between funding, policy, and vision.**

   Strongly agree: 14
   Somewhat agree: 7
   Weakly agree: 2

   Editor's note: The importance of this poll cannot be overstated. At the moment, the USG has not established a single agency or office that could oversee such a coupling, yet without it, the SUNS SMEs anticipate that we will be unable to answer key questions about NS&CI systems.

C.3 Poll Results: How the Nature of Funding Has Hampered and Harmed Our Efforts

This set of polls captures opinions related to the ways existing funding mechanisms for software analysis tools are sometimes at odds with the long-term development of software understanding capabilities.

1. **Due to the commonality across analysis applications, investment in reusable infrastructure components would give us a ___ return on investment (ROI) in the long-term (10+ years).**

   Poor ROI (< 1.0): 0
   Average ROI (~2.0): 0
   Good ROI (~5.0): 15
   Very Good ROI (~10.0): 10
   Exceptional ROI (~100.0+): 2

   Editor's note: A number of technical conversations during SUNs discussed various types of commonality across different software analysis scenarios. This poll was designed to ascertain just from a financial perspective how much value there may be in investing in reusable infrastructure components. Most of the SUNS SMEs were from mission-driven organizations; such organizations

historically have little interest in funding reusable infrastructure components though they have a great appetite in the types of answers that they may produce. Such organizations may find that a different investment strategy that aligns more closely to the nature of the technical problem is necessary to achieve significant mission impact. The result here is similar to Poll #4 in Section C.1, although this poll is more focused specifically on reusable software analysis infrastructure.

2. **When I work on a mission-funded project, I spend _________ of my time on infrastructure.**

   20% or less: 2
   20%-40%: 13
   40%-60%: 4
   60%+: 7

   Editor's note: This poll indicates that the majority of SUNS SMEs spend less than half their time on creating reusable analysis infrastructure or components. This is a symptom of the sharing and funding timeline problems, as discussed in Sections 7.7 and 7.9. The SUNS SMEs expect reusable infrastructure to be an important focus of tool development when working towards a broad, revolutionary software understanding capability.

3. **I have made bad design choices in my tools because of the need to deliver short-term mission results.**

   Yes: 25
   No: 2

   Editor's note: The nearly unanimous result for this poll is shocking, and points to a mismatch between best practices for long-term software understanding capabilities and best practices for immediate mission results. If we want to build long-term software understanding capabilities, this mismatch must be addressed.

4. **I have stopped working on a project before achieving impact because the funding ran out. With additional time, I am ____________ I could achieve much greater impact.**

   Confident: 10
   Somewhat confident: 10
   Not confident: 0
   N/A: 7

   Editor's note: Funding timelines frequently end software understanding related projects before they reach their intended impact, stressing the need to align realistic funding timelines with our project objectives.

C.4 Poll Results: Community

This poll is the only one SUNS SMEs developed to discuss the need for community building.

1. **The community is so fragmented that we operate in highly siloed groups such that we only make marginal progress in software understanding capabilities.**

   Agree: 23

   Editor's note: This fragmentation arises from many factors, including the sharing issues covered in Appendix C.5 and the non-technical issues addressed in many of the other straw poll questions. It can occasionally arise from inaction on the part of the SUNS SMEs themselves.

C.5 Poll Results: Sharing and Collaboration

This set of polls is about the need to share tools, research, and data, as well as the current state of sharing practices among the SUNS SME community.

1. **The process for asking permission to share is essentially a NO even if you know the answer is going to be YES.**

   Agree: 12
   Disagree: 15

   Editor's note: From a technical staff perspective, the bureaucratic process involved in making a request to share tools, source code, and software samples can at times be daunting. This poll was designed to understand whether the SUNS SMEs find, in practice, that the bureaucratic burden of sharing is so high that they don't even try to navigate the process. Although 55% of voting SMEs disagreed with this statement, the 44% who did is such a high percentage that it undoubtedly has a significant impact on the community.

2. **Not sharing has drastic, tangible impacts across our communities.**

   Agree: 27
   Disagree: 0

   Editor's note: Perhaps it goes without saying, but the "impacts" experienced by the SUNS SMEs are negative. Not being able to collaborate on research together or to share in-house developed tools and data sets means that we must all start nearly from scratch and cannot leverage each other's improvements. This results in significant duplicated effort and requires break-through innovations to be independently discovered multiple times. When the problem is as challenging as the development of software analysis capabilities, it requires a significant collaboration and team effort; although it has already been stated, it bears emphasizing that effectively isolating the pockets of USG expertise makes it effectively impossible to achieve the radical improvements in software understanding that the SMEs believe is possible.

3. **Not sharing is the equivalent of fraud, waste, and abuse.**

   Agree: 15
   Disagree: 12

   Editor's note: A majority of SUNS SMEs have encountered such bureaucratic difficulty in sharing research and tools that they feel the situation borders on being criminally wasteful.

4. **I have waited ____ for permission to be able to share a tool or result, slowing collaboration.**

   6+ weeks: 0
   6+ months: 10
   N/A (waiting for over 6 months and haven't been able to share yet): 17
   N/A (haven't had a sharing problem): 0

   Editor's note: Effective collaboration means sharing improvements on a daily or at least weekly pace. Having to wait months for permission to share technical advancements effectively kills collaboration. As an illustration, if people had to wait months to be able to fill their car's gas tanks, that would effectively kill everyone's ability to drive.

5. **The bureaucratic difficulties that exist with regards to sharing classified information often make it impossible to share even when there are no inherent reasons not to share.**

   Agree: 27
   Disagree: 0

   Editor's note: Sharing research, tools, and data in a classified environment has additional difficulties compared to unclassified sharing. The focus on obtaining prior approval and demonstrating a clear need-to-know makes perfect sense from the perspective of protecting classified information, but the extra burden makes timely sharing nigh-impossible, even when the other party is already known to have need-to-know and meet clearance requirements. A solution is needed to both respect classification concerns and radically reduce the sharing burden for trusted partners.

6. **Sharing leads to better outcomes. There is tremendous evidence in support of this conclusion and little evidence in support of the opposite.**

   Agree: 27
   Disagree: 0

   Editor's note: Complex research involves leveraging many different results and building upon previous research and tools through sharing to succeed. Current limitations on sharing prevent multiple software understanding research groups to share this research load.

7. **Open sourcing our tools accelerates our ability to have new staff "hit the ground running".**

   Agree: 27
   Disagree: 0

   Editor's note: A hidden benefit of open sourcing our tools is that the potential field of candidates for new staff can gain expertise in our tools before even being hired.

8. **Open sourcing our tools should be the default. If it cannot be open source, it should be born "government open source" (i.e., freely sharable within the government).**

   Agree: 25
   Disagree: 2

   Editor's note: While it is currently possible to open source tools developed by the government, the overwhelming majority of SUNS SMEs find the process too slow and onerous to be exercised regularly. Making open source (or government open source) a default position is a potential way to make open sourcing of government tools more viable. The value in open sourcing tools can be seen from the answers to Polls #6, #7, and #10 in this Section.

9. **If you don't want a project to be fully open, it is essential that you carve up the project to facilitate as much fully open sharing as possible.**

   Agree: 23
   Disagree: 4

   Editor's note: One potential concern regarding the open sourcing of software understanding tools in NS&CI is the potential risk to mission of doing so. If an adversary knows the full, precise technical details of the technical analysis of a particular software artifact, that adversary may be

able to learn details about that software artifact that further their own goals and harm US interests. Implicit in this question is the possibility of dividing such a technical analysis tool up into parts, where the non-sensitive parts are made open source, and the mission-specific or mission-sensitive aspects are protected within the government and not released to the public. Although there was no poll at SUNS specifically focused on this, it is the opinion of the editors that in many cases it may be as much as 95% of the tool that is mission agnostic and would not harm US interests to release (and would in fact advance US interests by enabling sharing, cultivate useful research in academia, etc.), while the remaining 5% of the tool contains sensitive, mission-specific information which should be withheld from release to protect US interests.

Presumed in this poll is the fact that architecting a tool to enable the public release of the non-sensitive portions while not releasing the sensitive portions requires time and effort. There is long-term value of this effort, but as discussed in Section 7.7, individual project funding is frequently inadequate to deliver the hoped-for mission impacts and is certainly insufficient to support the software-engineering efforts needed to maximize the benefit of sharing.

10. **People are more likely to use a crummy open-source tool than they are to use a highly capable, closely held (yet available) tool.**

    Agree: 20
    Disagree: 7

    Editor's note: This poll is perhaps surprising. There is value in a tool being highly capable; there is value in sharing. This suggests that the value in having the ability to share a tool with others or to learn about a tool using an internet search can in many cases outweigh the advanced capabilities already available today. This speaks to the vital nature and potential dramatic benefit of government stakeholders being aggressive about resolving the sharing barriers within the SUNS community.

11. **As a developer, knowing that I'm going to share the tool I develop with others changes my behavior, resulting in much higher quality tools and better overall engineering.**

    Agree: 15
    Disagree: 12

    Editor's note: This poll is designed to ask whether increased sharing of tools leads to higher quality tool development. Although the results are mixed, the majority of SMEs do believe an intent to share does improve tool quality.

12. **In this field, collaboration is essential for progress and sharing is the bedrock of collaboration.**

    Agree: 27
    Disagree: 0

    Editor's note: Sharing and collaboration are not the same thing. Sharing speaks to one party giving what they have already developed to another party. Collaboration speaks to working closely together to jointly create the thing in the first place. Sharing is a weaker form of interaction than collaboration. If you cannot share, you cannot collaborate. This poll reveals the SUNS SMEs belief that collaboration among pockets of software understanding SMEs within and beyond the government is essential for progress in this area. Government actions taken to improve sharing should have the aspirational goal of zero-friction collaboration for non-mission-sensitive software in this space.

C.6 Poll Results: Challenge Problems, Benchmarks, Competitions

This set of polls relates to the idea of creating challenge problems, datasets, and benchmarks along with a related competition each year to drive research. Inspiration for this idea comes from a variety of places, but a notable one in the computer science spaces is the SV-COMP yearly competition of software verification tools[130]. This competition, along with its benchmarks and data sets, has become a driving force in academic and industrial innovation, as most tools publish their results against this data set. By curating the types of problems represented in the dataset, the SV-COMP activity has highlighted certain types of problems and shaped the direction of the field. Software understanding activities could benefit from a similar construct.

1. **We need challenge problems and measurable benchmarks in this community. Additionally, we need data sets and metrics to drive and observe progress (e.g., something like SPEC but for software understanding).**

   Agree: 28
   Disagree: 0

   Editor's note: The Standard Performance Evaluation Corporation (SPEC) is a non-profit consortium that establishes, maintains and endorses standardized benchmarks and tools to evaluate performance for computing systems[141]. Such benchmarks and evaluations drive research and innovation toward particular goals. SPEC serves as a useful model for benchmark standardization. This poll demonstrates that the SUNS SMEs believe that there is a gap regarding having such benchmarks for software understanding and that filling this gap would have impact on the community.

2. **A suitable set of challenge problems and benchmarks and any associated organized competitions will encourage inter-organizational sharing, provide measurable progress indications, encourage development of usable tool frameworks, generate reusable artifacts, and grow an expert external community and incoming workforce.**

   Agree: 28
   Disagree: 0

   Editor's note: This poll provides more detail on exactly what benefits the SUNS SMEs believe we could derive from well-crafted benchmarks and evaluations in software understanding.

3. **Outputs and rules of engagement for the competition must encourage our goals of collaboration, integration, and progressing the science of software understanding — without losing the benefits of competition. The competition must publish each tool along with its competitive result, and reward: incremental progress, building on existing infrastructure, interoperability, and collaboration.**

   Agree: 28
   Disagree: 0

   Editor's note: Transparency in the competitive environment is necessary to achieve the benefits the community desires from benchmarks and challenge problems.

4. **Should the competition teams be required to make their tool open source and available for inspection?**

---

[141] See Standard Performance Evaluation Corporation, https://www.spec.org/.

Agree: 23
Disagree: 3
Abstain: 2

Editor's note: The SUNS SMEs noted that, in their experience with previous competitions in similar software spaces, a requirement to publish the competing tools and associated artifacts has helped to spur learning, innovation, and continued research after the event ends.

5. **Should the competition teams be required to provide an artifact that someone else can run?**

   Agree: 27
   Disagree: 1
   Abstain: 0

   Editor's note: See editor's note for Poll #4 above.

6. **Should the competition teams be required to write a detailed whitepaper?**

   Agree: 17
   Disagree: 6
   Abstain: 3

   Editor's note: See editor's note for Poll #4 above.

7. **This community needs to be funded to design these challenge problems and benchmarks, helping to ensure that the design achieves the goals.**

   Agree: 27
   Disagree: 1

   Editor's note: This poll demonstrates the need for funding organizations to materially support the development and execution of benchmark and challenge problem. Quality challenge problems and benchmarks suitable for driving research and innovation are unlikely to arise organically the way private sector competitions have.

8. **Government sponsors should not only fund the creation of these structured, motivating exercises and data sets, but also the execution of the competitions.**

   Agree: 27
   Disagree: 1

   Editor's note: See editor's note for Poll #7 above.

9. **Should the entire world be able to access these challenge problems and benchmarks?**

   Agree: 8
   Disagree: 8
   Unsure: 7

   Editor's note: The split vote is likely driven by the uncertainty of the content of the challenge problems and the potential impact to mission if sensitive problems are included. There are likely two broad perspectives supported by the SUNS SMEs – (1) that the competition should be held in the open and include academic and industrial participation, both foreign and domestic (as the SV-COMP competition and other academic-sponsored competitions are), and (2) that the competition should be held at the OUO/CUI level and restricted appropriately. As the content of the challenge problems and data sets was not discussed in detail, this was ambiguous when the question was

posed. It is likely that were a decision to be made regarding the level of sensitivity to include in the challenge problems and benchmarks, this split vote would largely be resolved.

10. Which do you care more about in the next 5 years:

A DARPA-level problem: 1
A challenge problem: 10
A puzzle: 1
A benchmark: 12

Editor's note: During the polling of this question, many participants noted that they wanted to vote for more than one option.

11. In the challenge problems, benchmarks, and competitions, would you care more about:

A foundational research question (that can be modified as we go along): 14
A mission-based question (e.g., does the SBOM describe the executable): 6
Abstain: 4

Editor's note: This poll perhaps reflects the sentiment reflected in other poll results and held fairly broadly by the SUNS SMEs that a premature attempt to achieve mission impact causes necessary foundational activities to be cut short and undermines the community's ability to achieve mission impact in the end. It perhaps also reflects the sentiment that selecting mission-based questions may narrow the ability of many groups to participate and that broad participation is to be preferred.

12. Every challenge problem we issue must include real-world examples. The benchmarks and data sets must include ________________ real-world examples as compared to constructed examples or toy examples.

75-100%: 2
50-75%: 7
25-50%: 7
0-25%: 5

Editor's note: The split vote for this question is likely the result of considering different stages of capability development. For some nascent software understanding challenges, toy examples are the most the current capability can handle. For more advanced software understanding capabilities, real-world examples are conceivable and would provide better tie to mission.