# *Software Understanding for National Security (SUNS)*

## *Technical Research, Development, and Engineering Roadmap*

Douglas Ghormley
Tod Amon
Christopher Harrison
Tim Loffredo

*Sandia National Laboratories*

December 10, 2024

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

## TABLE OF CONTENTS


LIST OF FIGURES ....... 4
DOCUMENT REVISIONS ....... 4
ACKNOWLEDGEMENTS ....... 5
1 INTRODUCTION ....... 6
1.1 Document Purpose and Scope ....... 6
1.2 Alignment to Related National Documents ....... 8
1.3 The National Challenge ....... 9
1.4 Terminology ....... 11
2 BACKGROUND AND RATIONALE ....... 12
2.1 Background and Rationale, BR1: Software Security vs. Understanding ....... 12
2.2 Background and Rationale, BR2: Software Understanding as a Chain Link Problem ....... 12
2.3 Background and Rationale, BR3: An Analogy between Physical Sciences and Software Understanding ....... 13
3 FOUNDATIONAL ASSUMPTIONS ....... 15
3.1 Foundational Assumption, FA1: Mathematical Modeling ....... 15
3.2 Foundational Assumption, FA2: Composability ....... 17
4 CROSS-CUTTING APPROACHES ....... 18
4.1 Cross-Cutting Approach, CC1: Dynamic and Static Analysis ....... 19
4.2 Cross-Cutting Approach, CC2: Machine Learning, Artificial Intelligence, and Large Language Models ....... 19
4.3 Cross-Cutting Approach, CC3: Human Factors and Human Studies ....... 22
4.4 Cross-Cutting Approach, CC4: Design for Introspection and Debugging ....... 24
5 RESEARCH, DEVELOPMENT, AND ENGINEERING ROADMAP ....... 25
5.1 R&D Challenge, R1: Formal Foundations for Software Reasoning ....... 28
5.1.1 R1.1 Reasoning logics ....... 29
5.1.2 R1.2 Approaches for Guiding Precision vs. Abstraction Tradeoffs ....... 30
5.1.3 R1.3 Harnessing Modern and Emerging Hardware Architectures ....... 30
5.1.4 R1.4 Compositional Reasoning ....... 31
5.1.5 R1.5 Machine Learning (ML) Approaches to Reasoning Systems ....... 32
5.2 R&D Challenge, R2: Analysis Architectures and Automated Tool Synthesis ....... 33
5.2.1 R2.1 Analysis Structures and Techniques ....... 34
5.2.2 R2.2 Composable Analysis Architectures ....... 35
5.2.3 R2.3 Automated Analysis Tool Synthesis ....... 37
5.2.4 R2.4 Designing for Human-in-the-Loop Analysis ....... 38
5.2.5 R2.5 Analysis Search Strategies ....... 39
5.2.6 R2.6 Adequate Foundational Tooling for All Target Binaries ....... 40
5.3 R&D Challenge, R3: Software Execution Modeling ....... 40
5.3.1 R3.1 CPU Instruction Modeling ....... 41
5.3.2 R3.2 Hardware Peripheral Modeling ....... 42
5.3.3 R3.3 Initial State Modeling ....... 42
5.3.4 R3.4 Memory Use Modeling ....... 43
5.3.5 R3.5 Indirect, Interrupt, and Exceptional Control Flow Modeling ....... 44
5.3.6 R3.6 Operating System Interaction Modeling ....... 45
5.3.7 R3.7 External Library Call Modeling ....... 45
5.3.8 R3.8 Execution Concurrency Modeling ....... 46
5.3.9 R3.9 Sandbox Environment Modeling ....... 46
5.4 R&D Challenge, R4: Model Generation Techniques ....... 47
5.4.1 R4.1 Rapid Model Generation and Validation ....... 47
5.4.2 R4.2 Modeling Refinement Strategies ....... 48
5.4.3 R4.3 Model Inference for Missing Components ....... 49
5.5 R&D Challenge, R5: Analysis Tool Ecosystem ....... 49
5.5.1 R5.1 Analysis Tool Characterization ....... 52

5.5.2 R5.2 Tool Configurability, Interfaces, and Interoperability ..... 53
5.5.3 R5.3 Program Knowledge Store ..... 54
5.5.4 R5.4 Automated Analysis Campaign Orchestration ..... 55
5.6 R&D Challenge, R6: Semantic Knowledge Inference ..... 56
5.6.1 R6.1: Computational Heuristics for Semantic Inference ..... 56
5.6.2 R6.2: Human Factors Analysis of Manual Reverse Engineering ..... 57
5.6.3 R6.3: Semantic Inference Through Machine Learning Techniques ..... 58
5.7 R&D Challenge, R7: Hierarchical Question Decomposition and Evidence Composition ..... 58
5.7.1 R7.1: Human Factors Studies and Cognitive Mapping ..... 60
5.7.2 R7.2: Applications from Formal Software Verification ..... 61
5.7.3 R7.3: Analysis Evidence and Provenance Collection ..... 62
5.7.4 R7.4: Analysis Confidence Under Uncertainty ..... 62
5.7.5 R7.5: Artificial Intelligence Approaches to HQD and EPC ..... 63
5.8 R&D Challenge, R8: Datasets, Benchmarks and Ground Truth ..... 63
5.8.1 R8.1 Ground Truth Representation Standards ..... 65
5.8.2 R8.2 High-level Datasets and Benchmarks ..... 66
5.8.3 R8.3 Low-level Datasets and Synthetic Benchmarks ..... 66
5.8.4 R8.4 Dataset Repository ..... 67

**6 DISCUSSION ON PRIORITIZATION AND SEQUENCING ..... 68**
6.1 Validate the Composable Modeling Hypothesis ..... 69
6.2 Start with the Foundations ..... 69
6.3 Crawl, Walk, Run ..... 71

**7 SUMMARY AND CONCLUSIONS ..... 72**

## LIST OF FIGURES

Figure 1: Visual summary of the technical topics is in this roadmap. .......... 7
Figure 2: Scale of the Software Understanding Problem. .......... 10
Figure 3: Notional overview of a software understanding campaign. .......... 27
Figure 4: Conceptual relationship of R1-R5 in this roadmap. .......... 28
Figure 5: Notional depiction of a monolithic tool. .......... 35
Figure 6: A notional example of the full, end-to-end process of software understanding. .......... 59
Figure 7: Notional example of a partial Hierarchical Question Decomposition (HQD). .......... 60

## DOCUMENT REVISIONS

*Table 1: Document Revisions*

| Revision Date | Revision Description |
|---|---|
| Dec. 10, 2024 | Initial publication |

## ACKNOWLEDGEMENTS

Many people provided excellent input and feedback in drafting this roadmap. The authors would like to thank the following people (listed alphabetically) for their input: Denis Bueno, Karin Butler, Todd Fine, Scott Heidbrink, Samuel Mulder, Geoff Reedy, Katie Rodhouse, and Ryan Vrecenar.

In addition to these individuals, we would like to acknowledge the intellectual discussions with the subject matter experts at SUNS 2023 which helped inform the authors' thinking on software understanding.

# 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 recent decades, software has been integrated into every facet of government and society, including NS&CI missions. Our nation's historical choices on how to leverage software widely have resulted in economic opportunity and prosperity, but also a tremendous gap between our total dependence on software for NS&CI missions and our extremely limited capability to understand and validate that software.

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 are 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 currently exist. The broad set of mission questions that need to be addressed, the vast array of software upon which we depend, and the speed at which answers are needed, are all challenging requirements of the software understanding problem that exceed current technical capabilities.

Every U.S. government (USG) agency faces this same challenge. 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 across the government 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.

Establishing high confidence in the software that underpins the nation's NS&CI systems requires empowering mission owners to pose mission-specific questions about that software's possible behavior and to gather technical evidence packages about that behavior to inform risk mitigate and acceptance decisions. Building the necessary technical analysis capabilities will require a significant, coordinated, national-level effort.

This document serves as a roadmap to guide research, development, and engineering investments toward the technical tools and capabilities needed to create those evidence packages, enabling mission owners to adequately understand the software upon which our NS&CI missions critically depend.

## 1.1 Document Purpose and Scope

The purpose of this document is to present a technical research, development, and engineering roadmap toward the capabilities that will enable the U.S. government to achieve greater understanding of software behavior within NS&CI mission spaces. The authors acknowledge that this roadmap is based on particular foundational assumptions and our perspective about likely viable solutions. Though this perspective is informed by extensive familiarity with the academic literature and years of direct research experience, we fully expect that this roadmap will need to

change in the future—perhaps radically—as more is learned and as technology advances. Thus, we view this roadmap is an *initial draft*.

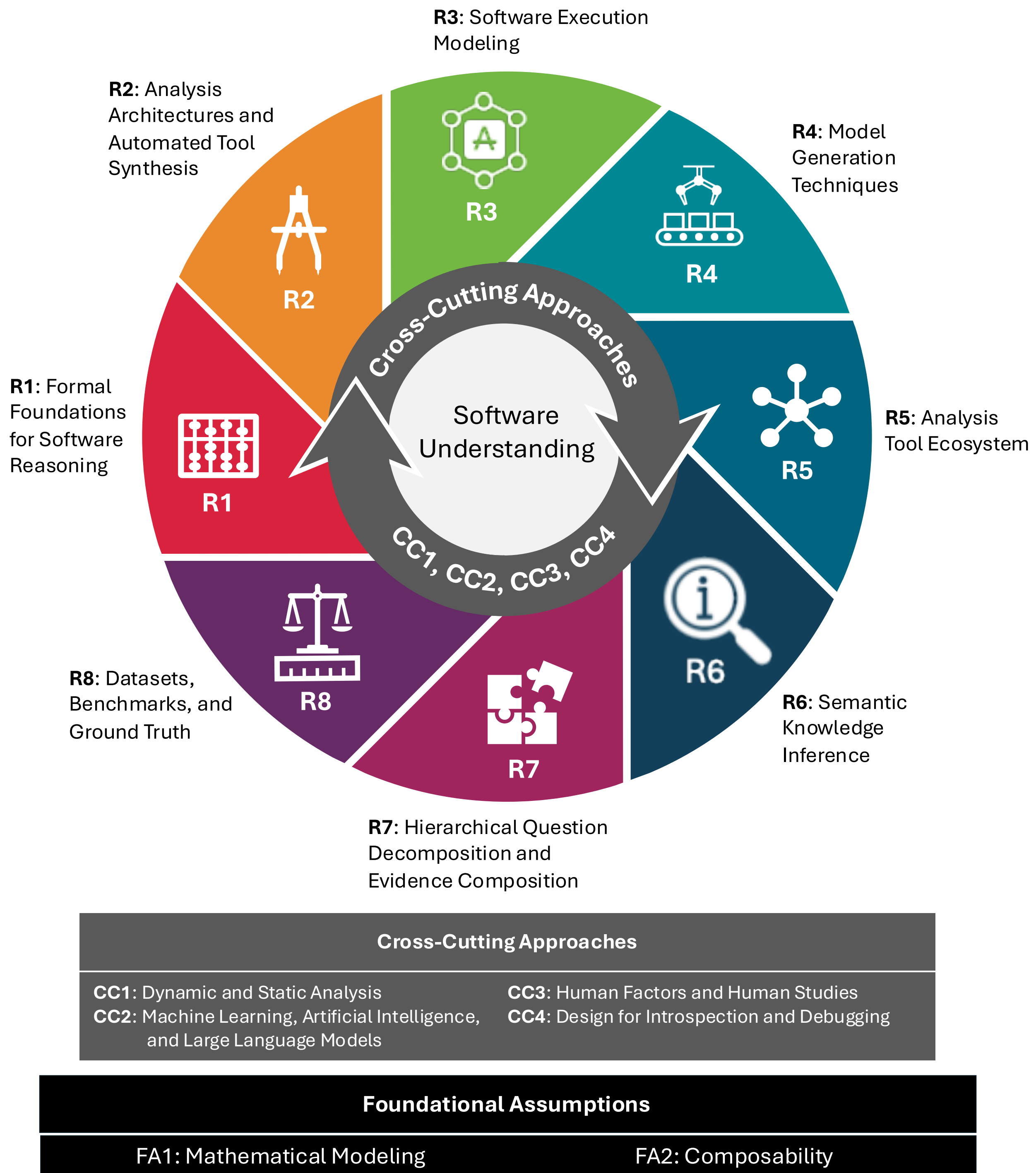


*Figure 1: Visual summary of the technical topics is in this roadmap.*

A secondary purpose of this roadmap is to inform policy makers, program managers, and other high-level decision makers as to the scale of the research, development, and engineering needed to create the address the software understanding needs of NS&CI missions.

The scope of the document includes the following:

- Background and rationale, explaining some of the reasoning behind the roadmap (Section 2)
- Foundational assumptions that underpin the roadmap (Section 3)
- Cross-cutting approaches that span multiple research areas of the roadmap (Section 4)
- The research thrusts and sub-thrusts of the roadmap (Section 5)
- A discussion of the prioritization and sequencing for pursing the roadmap (Section 6)

Figure 1 presents a visual overview of the technical elements discussed in the roadmap, including the research thrusts (R#), Cross-Cutting Approaches (CC#), and Foundational Assumptions (FA#).

The following topics are out of scope for this document:

- Roles and responsibilities
- Timeframes and resourcing estimates for achieving roadmap goals
- Strategies for cultivating and growing the technical community necessary to execute this roadmap
- Policy impediments that currently prevent the existing pockets of community with technical interest in this problem domain from coming together organically

This document is a companion to "The National Need for Software Understanding"[1]. For a thorough explanation of the software understanding problem, root causes, and needs, see that report.

## 1.2 Alignment to Related National Documents

This document expands upon several national level strategies, implementation plans, and other roadmaps.

- Software understanding is a key capability needed to meet the national emergency declared in Executive Order 13873, "that foreign adversaries are increasingly creating and exploiting vulnerabilities in information and communications technology and services, which store and communicate vast amounts of sensitive information, facilitate the digital economy, and support critical infrastructure and vital emergency services, in order to commit malicious cyber-enabled actions, including economic and industrial espionage against the United States and its people."[2]
- Software understanding is a necessary aspect of "Enhancing Software Supply Chain Security" as called for in Executive Order 14028[3] to provide the technical insight and evidence needed to implement "more rigorous and predictable mechanisms for ensuring that products function securely, and as intended".

[1] D. Ghormley, T. Amon, C. Harrison, and T. Loffredo, "The National Need for Software Understanding", Sandia National Laboratories, Jan 17, 2024.
[2] United States, Executive Office of the President. Executive Order #13873: Securing the Information and Communications Technology and Services Supply Chain. May 15, 2019. *Federal Register*, vol. 84, no. 96, pp. 22689-22692, https://www.govinfo.gov/content/pkg/FR-2019-05-17/pdf/2019-10538.pdf.
[3] United States, Executive Office of the President. Executive Order #14028: Improving the Nation's Cybersecurity. May 12, 2021. *Federal Register*, vol. 86, no. 93, pp. 26633-26647, https://www.govinfo.gov/content/pkg/FR-2021-05-17/pdf/2021-10460.pdf.

- Furthermore, software understanding is key to performing many types of independent validation of the Software Bills of Material (SBOMs) required by Executive Order 14028[3] and included in CISA's Open Source Software Security Roadmap[4].
- Federally mandated Zero Trust Architecture (ZTA) adoption is intended to defend against increasingly sophisticated and persistent threat campaigns[5]. In light of the repeated infiltration of software supply chains by such campaigns, software understanding is necessary to analyze the software that underpins ZTAs to establish its trustworthiness through technical analysis rather than presumption or developer attestation[6].
- Software understanding is necessary to achieve the goals of the National Cybersecurity Strategy, in particular Strategic Objective 5.5: Secure Global Supply Chains for Information, Communications, and Operational Technology Products and Services[7].
- "Software understanding is a vital aspect of CISA's Secure By Design strategy"[8].

## 1.3 The National Challenge

The scope of the national challenge that this roadmap is designed to address is described in detail in the report "The National Need for Software Understanding"[9]. As a reminder, Figure 2 depicts the scale of the Software Understanding problem as contemplated by this roadmap. Although point solutions may exist for certain combinations of questions and systems, this roadmap seeks approaches capable of scaling to the scope of the national need (see Section FA1 for more information). This roadmap also favors the use of mathematically precise and rigorous approaches where feasible, although other approaches can be useful, or even essential, for progress on some technical challenges.

[4] Cybersecurity and Infrastructure Security Agency, "CISA Open Source Software Security Roadmap." September 2023. https://www.cisa.gov/sites/default/files/2024-02/CISA-Open-Source-Software-Security-Roadmap-508c.pdf.

[5] Shalanda Young, Office of Management and Budget, Executive Office of the President. "Moving the U.S. Government Toward Zero Trust Cybersecurity Principles." https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf

[6] See https://www.gao.gov/products/gao-22-104746, https://www.gao.gov/assets/d24105658.pdf and https://www.gao.gov/assets/gao-24-107231.pdf

[7] https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf, see also https://www.whitehouse.gov/wp-content/uploads/2024/05/National-Cybersecurity-Strategy-Implementation-Plan-Version-2.pdf

[8] Cybersecurity & Infrastructure Security Agency, "CISA's Efforts Towards Software Understanding", April 26, 2024. https://www.cisa.gov/news-events/news/cisas-efforts-towards-software-understanding

[9] D Ghormley, T. Amon, C. Harrison, and T. Loffredo, "The National Need for Software Understanding", Sandia National Laboratories, Jan 17, 2024.

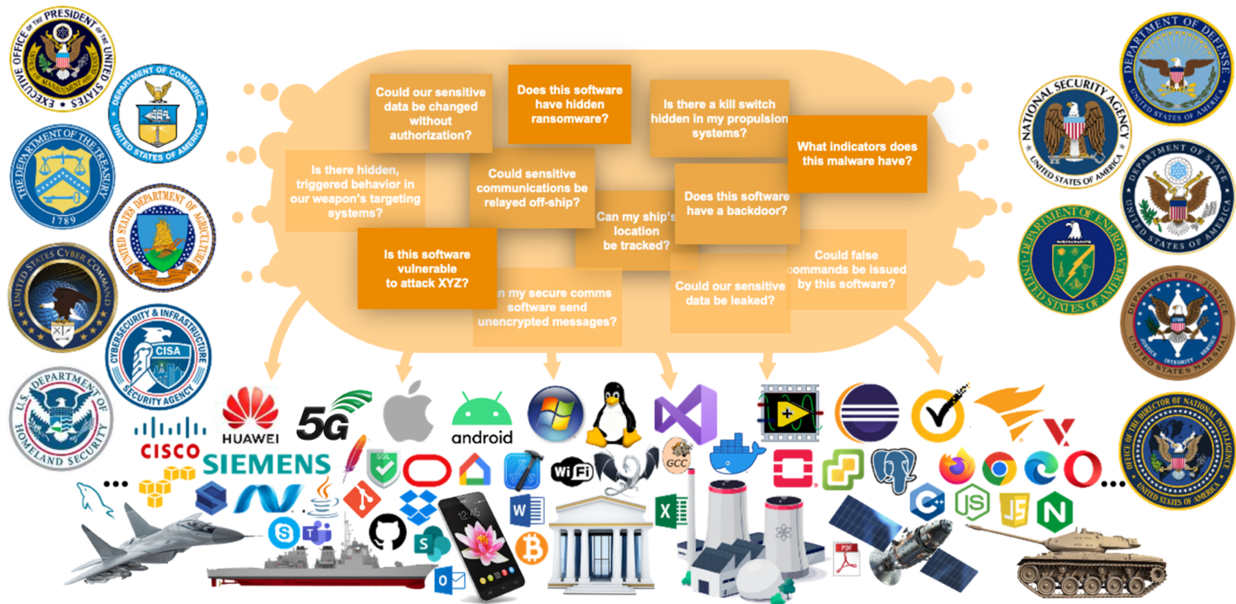


*Figure 2: Scale of the Software Understanding Problem.*
*At the bottom of the diagram are notional examples of the nation's critical infrastructure and national security physical systems. Just above those are notional examples of various software used in the design, development, deployment, maintenance, and operation of those systems. Flanking on the left and right are seals of various departments and agencies in the USG which own NS&CI missions. In orange in the center are notional questions that these agencies would ideally be able to answer questions about their software to have high confidence in their missions.*

There are a variety of different types of software in scope for this roadmap, including:

- Enterprise software, such as desktop operating systems and applications, web servers and services, cloud computing and end point security software, and more.
- Enterprise and Internet network software, including network backbone software, boundary protection and filtering software, secure communications software, server software, network monitoring and administration software, etc.
- Mobile device operating systems and applications on end-point devices as well as the providers' core network and core services.
- Industrial control system (ICS) software, Internet of Things (IoT) software, and other embedded software systems.
- Custom software used for defense applications, such as software used in military vehicles.
- Design tools, including compilers, linkers, computer aided design (CAD) tools, programming environments, integrated circuit board and chip design and layout tools, and anything else used in the forward engineering of civil structures, critical infrastructure, transportation systems, medical or life-safety equipment, digital hardware and software, etc.

## 1.4 Terminology

- **Software understanding**: the practice of understanding the intended and unintended functionality of a given software system. In most cases, especially in the presence of 3rd party software, understanding is possible only by analyzing the software artifact directly, rather than relying on proxies or substitutes such as attestation, documentation, mitigations, or policies and procedures. Understanding is most needed before software is placed into use so that undesirable behavior can be prevented**.** While software understanding can be manual, and manual analysis is still done today across many missions, most missions require significant degrees of automation. “Software understanding” is a superset of “software measurability” and “software measurement” as discussed in the “Back to the Building Blocks: A Path Toward Secure and Measurable Software”[10] and the “Federal Cybersecurity Research and Development Strategic Plan”[11]. “Software understanding” applies in both forward engineering contexts (e.g., during software development) as well as reverse engineering contexts (e.g., forensic software analysis).
- **Forward engineering:** the process of designing, developing, or creating new software. Forward engineering includes the development of any software development artifact such as design documents, requirements, or source code, not just the final software product or binary. Improvements in forward engineering processes can help to shape the development of new software prevent unintended behavior that could fail to meet safety and other mission requirements or otherwise threaten the mission.
- **Reverse engineering:** the process of analyzing, decomposing, and understanding existing software systems as delivered. Reverse engineering may include the attempted recreation or extraction of the design and structure of forward-engineered software systems. Improvements to reverse engineering processes can help in discovering and understanding the possible behaviors of existing software systems for various mission purposes. The scope of this roadmap is primarily focused on reverse engineering, not forward engineering, However, there is significant overlap in the goals, techniques, and research needed for both forward and reverse engineering, so some of the recommendations in this roadmap may apply to forward engineering as well.
- **Binaries**: a machine-readable version of a program designed for direct execution on a processor (e.g., x86 Windows PE file) or for interpretation by a virtual machine (e.g., Java bytecode), whether intended for use on a desktop, mobile, or embedded system. *Note: although many software artifacts are delivered and used as binaries, many NS&CI-relevant software artifacts are not—for example, interpreted Python code or bash scripts. These other kinds of software are still relevant to software understanding, so this roadmap is not limited to binaries only, though the authors will use that the term “binaries” as an informal short-hand for the broader set.*

[10] “Back to the Building Blocks.” *The White House*, February 2024, https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[11] “Federal Cybersecurity Research and Development Strategic Plan.” Cyber Security and Information Assurance Interagency Working Group, December 2023, https://www.whitehouse.gov/wp-content/uploads/2024/01/Federal-Cybersecurity-RD-Strategic-Plan-2023.pdf.

# 2 Background and Rationale

This section discusses some background and rationale which provide context for the roadmap below, in particular these three points:

- BR1: Software Security vs. Understanding
- BR2: Software Understanding as a Chain Link Problem
- BR3: A Useful Analogy between Physical Sciences and Software Understanding

Additional detail and rationale can be found in the report "The National Need for Software Understanding", specifically sections 2-4[9].

## 2.1 Background and Rationale, BR1: Software Security vs. Understanding

Software understanding includes more than just software security. Software security is concerned with confidentiality, integrity, and availability, as well as issues such as authentication, authorization, and vulnerabilities. Most commonly, software security discussions center around issues that command broad or universal agreement about their negative impact on software (i.e. memory safety issues, common vulnerability classes, etc.).

In contrast, software understanding is concerned with all of these, as well as a much broader range of questions, including bespoke, mission-specific questions that will never find universal consensus. For a more detailed list of example Mission Questions, see Appendix A of "The National Need for Software Understanding". All software mission questions require an analysis of the software to seek evidence to answer the mission question at hand, but many questions are not traditional software security questions; rather, they are about *understanding* a key aspect of software behavior to inform mission decisions. This roadmap seeks to elucidate the research needs that are common across many such areas of software inquiry.

One key issue worthy of special note is that there are circumstances in which disagreements arise regarding whether a particular behavior is a feature or a vulnerability. An example of this is the Log4j issue[12]. There are also mission-specific system behaviors that no security tool would consider a vulnerability (e.g., these two valves should never be open at the same time in this industrial control system).

## 2.2 Background and Rationale, BR2: Software Understanding as a Chain Link Problem

Software understanding is an example of a *chain link* problem. To analyze the possible behavior of a software binary, an analysis tool must first model the system and the software. Poor models result in poor analysis results, whether incomplete coverage, false positives, false negatives, inadequate precision, and more. An extensive set of models must work in concert to provide accurate, useful analysis results; models must capture relevant aspects of the initial program state, the semantics of the processor instructions, hardware peripherals, the memory usage patterns of the software, operating system calls, external library interactions, any concurrency or interrupts, remote network interactions, and more. Program control and data paths typically flow through many of these models and must be appropriately tracked to analyze the software's behavior. It is this flow of control and data through the collection of models that creates the chain link problem. When any

[12] See Log4j issue report at https://issues.apache.org/jira/browse/LOG4J2-313 (accessed June 19, 2024).

one of these pieces is weak, it can negatively affect the entire analysis and cause total failure through inaccuracy or failure to complete the analysis at all.

To illustrate, here are a few observations about chain link problems and how to apply those observations to software understanding:

- A chain has multiple links. In software understanding analysis tools, there are multiple algorithms, tasks, and subtasks that must all work in concert to be successful. Simpler tools may have a single element that can be optimized for one particular task, analogous to a hammer; but software understanding requires many moving parts to work in concert, analogous to clockwork.
- The chain is only as strong as the weakest link. When just one link is weak or flawed, the overall system is weakened or flawed.
- Strengthening one link in the chain typically doesn't make the chain stronger. In the inter-related links needed for software understanding, improvements to one part of the analysis are often not reflected in an overall improvement of the chain—unless that one part happened to be weak link limiting the overall analysis.

In software understanding, there are some additional observations that inform this roadmap:

- In software understanding for third party systems, most links are weak – the overall ability to analyze software executables is very poor relative to the broad need.
- In software understanding, modern software analysis experts *currently cannot measure individual links* to see which are strong and which are weak – that is, in this chain link problem of software understanding, no one can currently identify which links are the limiting factor for any given analysis challenge and no one can easily measure incremental progress on individual problems.
- In software understanding, many tools are monolithic, not composed from easily reusable components. Consequently, the "links" of the software understanding chain are closely intertwined within a monolithic tool, making it challenging to even identify what a separate link may be.
- In software understanding, different tool elements can have different sensitivities. Some elements of software understanding tools are fundamental with no sensitivities whatsoever, such as the technique of *abstract interpretation* or tools to resolve *satisfiability* (SAT) problems, which have been widely developed in both academia and industry for decades. Other elements of software understanding tools may be very particular to sensitive aspects of NS&CI systems or mission questions. Software understanding solutions should be pursued in such a way that foundational, non-sensitive techniques and solutions can be shared broadly, while mission sensitive elements can be restricted to appropriate parties.

## 2.3 Background and Rationale, BR3: An Analogy between Physical Sciences and Software Understanding

For many, it is difficult to understand the challenge of analyzing software. One analogy that may be useful comes from biochemistry. For centuries, scientists studied biological organisms at a high level, learning many useful things. The invention of the first microscopes and discovery of cells unlocked a new, deeper level of insight which led to additional understanding. Prior to this invention, we understood many things, but that understanding was limited in the early years by gaps in our knowledge that we often didn't know existed. In recent decades, a series of breakthroughs have enabled scientists to analyze the low-level foundations of human biochemistry

in ways unimagined by previous generations, ushering in a revolution of understanding of metabolic pathways.[13]

To achieve these unprecedented levels of understanding, the scientific community over the years first had to understand atoms with their electron shells, charges, number of protons, atomic weight, and more. But that alone was not enough. They also needed to understand how those atoms combine into molecules, types of bonds, molecular geometry, bond angles, molecular polarity, exothermic and endothermic reactions, and more. But that was still not enough. Biochemistry studied particular families of molecules such as enzymes, nucleic acids, carbohydrates, enzyme kinetics, and molecular signaling cascades. The community further need to understand DNA, RNA and the mechanisms that produce various proteins and how those proteins interact to create chemistry-based machines that underpin all of life. Once the right foundational prerequisites were in place, the scientific community achieved the ability to reason about metabolic pathways in ways previously unimaginable.

This illustration describes a series of layers, from low-level to high-level, with higher levels dependent on lower levels. Questions regarding high-level layers often requires understanding the lower layers out of which the higher layers are constructed. But each layer adds new behaviors that are not present in the lower layers.

Software is similar. In software, 1's and 0's comprise the lowest layer. These are collected into "bytes" which are then grouped together to form executable instructions, which are the atomic unit of software execution (similar to the physical atoms of chemistry in this analogy). These instructions combine into sequences called basic blocks (analogous to molecules in chemistry). These basic blocks are combined into functions, which interact with each other in a variety of ways (analogous to proteins in biochemistry). A program (an organism in the analogy) is made up of functions, ranging from just one in simplistic cases to hundreds of thousands in more typical computer or mobile software.

Just as the physical and biological sciences need different scientific equipment to measure and characterize atoms, molecules, proteins, and metabolic pathways, so software understanding needs different algorithms and analysis techniques to characterize the various layers in software. In the physical sciences, the simple microscope alone was not enough to meet all measurement needs, but scientists over the decade invented spectroscopy, chromatography, magnetic resonance (used in an MRI), electrochemical measurement techniques, electron microscopes, and much more. In software analysis, there are analysis techniques today at various levels in the software stack, but these collectively fall far short of what's needed to meet the national scale needs in understanding with high rigor the software that underpins national security and critical infrastructure missions.

The nation needs to invent the software equivalent of the electron microscope, spectroscopy, chromatography, as well as the decision-making frameworks for understanding the entire system.

This means that in any roadmap, there will be themes that show up at multiple layers of analysis. For example, the need to measure, the need to model, the need to analyze behavior, the need for the human to introspect the system, the need to pose questions and seek evidence, the need for scalability, the need for modular composition, all apply at many layers, from low-level to high-level.

---

[13] Stanford Medicine, "Pathways of Human Metabolism Map", https://metabolicpathways.stanford.edu/ (accessed June 19, 2024).

# 3 Foundational Assumptions

Practically speaking, the scale of the national need depicted in Figure 2 requires widespread reuse of components to have a reasonable return on investment (ROI). To accomplish this, the authors have developed this roadmap around two foundational assumptions to enable scalable and reusable capabilities. As mentioned in Section 6, validating these foundational assumptions should be a priority, with the lessons learned used to produce a new version of this roadmap.

## 3.1 Foundational Assumption, FA1: Mathematical Modeling

Modern software systems have an exponential number of states with respect to the size of the program—vastly more states than the number of atoms in the universe, for any moderately sized modern program[14]. In principle, mission-threatening behaviors could lurk anywhere in this complete state space of the software. Exploring and understanding every state in this set in detail is not feasible, but fortunately, to answer a given mission question, it's typically necessary to only examine a subset of the states. For this to be practical, the analysis must aggressively abstract away detail not relevant to the question, while retaining precision on the aspects of the software's behavior necessary to answer the question. The most promising approach to answering questions about software through abstraction and reasoning is through mathematical modeling.

Mathematical modeling in the context of software understanding involves creating abstract representations of a program's behavior using mathematical constructs such as equations, graphs, and logical formulas. This technique allows software analysts to reason about the program's properties without executing it, making it particularly powerful for analyzing large software systems. By translating code into a mathematical model, software analysts can apply various theoretical tools to gather the information they need to answer any given mission question about the software. For instance, control flow graphs can help analysts understand the possible execution paths, while data flow equations can track how data moves and transforms throughout the program. The abstraction provided by mathematical modeling not only simplifies the complexity inherent in large programs but also enables the use of automated tools to perform rigorous and scalable analysis, ensuring that even subtle and hard-to-detect issues can be identified and addressed efficiently.

There are other approaches to solving this software understanding conundrum besides mathematical modeling—for example, dynamic testing, using proxies for software understanding, or manual human analysis—but those other approaches are not capable of providing the technically-based, rigorous, reliable, rapid, and repeatable software understanding capabilities the nation needs (for more discussion on this, see Section 3.3, "The Current State of Software Understanding Alternatives" in "The National Need for Software Understanding"[1]). As a result, the core approach to software understanding advocated in this roadmap is based on mathematical modeling strategies. If composable (see FA2 below) modeling approaches cannot be identified that suit the purpose this roadmap, the research thrusts within this roadmap should be revisited to reflect a new set of fundamental assumptions.

An important key observation about mathematical modeling, and software analysis in general, is that no one software analysis tool is adequate for all software analysis tasks. Given a class of

[14] Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P. (2012). Model Checking and the State Explosion Problem. In: Meyer, B., Nordio, M. (eds) Tools for Practical Software Verification. LASER 2011. Lecture Notes in Computer Science, vol 7682. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35746-6_1

software programs or systems under test (P), specific information to gather about those programs (related to Q), and a particular set of analysis resources to use (R), to be effective, an analysis tool (A) must be generated *for this specific combination of P, Q, and R.* That is, the analyzer is a function of P, Q, R.

$$A = f(P, Q, R)$$

To illustrate this with an example, consider two related scenarios: (1) an individual user context in which an analysis tool that analyzes a program on a laptop when the user double-clicks to run it, and (2) a national security mission context in which mission owners desire a high-level of confidence that critical software is free from advanced persistent threat (APT) malicious additions. In the former scenario, the analysis is expected to produce an answer about standard malware threats within seconds on modest hardware. In the latter scenario, the mission owner may choose to analyze key software for weeks in a thousand-core cloud system prior to placing it into operational use.

In this example, one can see that the types of questions that a typical consumer has may be very different than a national-security mission owner—thus the Q may be different. Even if the question is the same, the resources (R) dictate radically different algorithms, approaches, confidence thresholds, and more. And even if both Q and R are the same, the analysis tools needed to analyze specific classes of programs (P) can be wildly different—tools to analyze a typical desktop application are unlikely to work well (or at all) for industrial control systems or flight control software.

Software understanding is not one problem, but a vast space of problem instances. Successful analyses (A) incorporate carefully considered tradeoffs tuned to the specific P, Q, and R. Consequently, analysis tools do not generalize well to other classes of programs, mission questions, or resource constraints; not because of implementation quality issues but because of fundamental characteristics of software and theoretical limits on analyzability.

This observation has at least three key implications:

1) **Need for Reuse:** The analysis tool, A, when sufficiently customized to be effective at answering a particular mission question on a certain family of software binaries, has very limited reuse potential as a unit. However, the cost to the USG of fabricating a custom analysis tool from scratch for every combination of P, Q, and R is untenable. The USG needs a practical path forward for creating the set of tools capable of meeting the broad national need depicted in Figure 2. One promising approach is to create an ecosystem of reusable, configurable tools and components out of which any given specific analysis tool can be constructed, aggressively minimizing over time the amount of custom, manual work that must be done[15].
2) **Need for Iteration:** For a given P, Q, and R, it is unlikely that the analysis tool can be constructed correctly *a priori*. Even deciding in advance what models with what precision are needed for the analysis is difficult and error-prone, requiring information about the program which requires analysis to determine, creating a classic "chicken-and-egg" conundrum. Thus, it seems likely that the analysis tool, A, will need to be constructed through an iterative process. This further suggests there is a space of options which is to be searched to find a tool, A, that is adequate to gather the necessary evidence to answer Q. In such cases, different search strategies will have different characteristics.

---

[15] There are a few tools today with some element of reusability that may provide starting points for such an ecosystem, but the available options collectively fall far short of what is needed.

3) **Need for Human Integration:** The process described in #2 will not be mature for some time. Today, analysis tools are typically built entirely through manual effort with no automation in the tool construction process. Immediate, full automation is unreasonable to expect. Consequently, the research roadmap should chart a path of hybrid human/machine partnership in constructing analysis tools, identifying strategies to empower the analysis tool writers to be orders of magnitude more efficient and empowering them to rapidly identify root causes of problems in tools under development.

## 3.2 Foundational Assumption, FA2: Composability

The scale of the software analysis challenge depicted in Figure 2 has implications for the approaches that should be explored to solve it. The incredibly diverse set of mission questions, families of software of interest, and resource tradeoffs result in a monumental challenge. Isolated or uncoordinated efforts to create analysis solutions for individual combinations of P, Q, and R (to use the terminology of FA1) have already result in duplicated efforts whose components and results are incompatible with other efforts. In the limit, fully addressing the national need depicted in Figure 2 would require developing a tool for each combination of P, Q, and R—an approach that is economically infeasible.

Consequently, it is an assumption of this Roadmap that an essential element to practical solutions must involve approaches that maximize reuse through *composability*[16]—at each step, individual components must be designed so that they can be reused and assembled into solutions appropriate to the specific problem at hand. It is likely not possible to achieve this aim perfectly, for example, necessitating some amount of custom development or configuration to tie a set of reusable components together. However, perfection is not necessary to reap enormous return on investment if significant reuse can be accomplished. The amount of customization should be aggressively minimized in execution of this roadmap.

Composability has been studied in a variety of domains, from modeling and simulation[17,18] to Web services[19] to satisfiability theories[20]. There are at least three facets of composability that this roadmap has in view.

1. **The composability of foundational logic systems.** Tolk and Maguira, speaking in the context of compositionality within modeling and simulation systems, address this point this way: "In order to achieve meaningful interoperability of simulation systems on the technical level, composability of the underlying conceptual models is a necessary requirement."[21] Thus, before any actual tool can be designed to be compositional, one must first ensure that the mathematical foundations used to model software semantics and support reasoning

---

[16] Although the terms "composability", "compositionality", or "combination" may have technical nuances in different domains of software understanding, in this roadmap they are used interchangeably in their informal senses.
[17] Paul K. Davis, Robert H. Anderson, "Improving the Composability of Department of Defense Models and Simulations". RAND National Defense Research Institute, 2003.
[18] C. Szabo and Y.M. Teo, "An Approach for Validation of Semantic Composability in Simulation Models", 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation.
[19] Brahim Medjahed and Athman Bouguettaya, "A Multilevel Composability Model for Semantic Web Services", IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 17, NO. 7, JULY 2005.
[20] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar, "A Tutorial on Satisfiability Modulo Theories (Invited Tutorial)". CAV 2007, Lecture Notes in Computer Science 4590, pp. 20–36, 2007.
[21] Andreas Tolk and James Maguira, "The Levels of Conceptual Interoperability Model", 2003 Fall Simulation Interoperability Workshop Orlando, Florida, September 2003.

about software behavior must be composable at the conceptual level. R1 addresses this foundational level of composability.

2. **The composability of analysis components and tools.** In addition to the conceptual, mathematical foundations discussed in #1, the analysis components and tools must be designed as ensembles of reusable parts. These concepts go beyond mere modularity. There are two parts to this: the technical composability problem, and the semantic composability problem[22]. The technical composability problem centers on interfaces, protocols, data structures and such to ensure that composability is possible. The semantic composability problem focuses on ensuring that the aggregate composition of components has the intended behavior, achieves the desired goal, and exhibits the desired emergent properties. Leveraging the conceptual foundations of R1, R2 addresses composability internal to a given analysis tool's architecture whose design must facilitate the composition of reusable components, R3 deals with the models of software execution that will be leveraged by the tools and that themselves must be designed for both technical and semantic composability, while R5 addresses the composability across an ecosystem of analysis tools, each interacting in an iterative analysis campaign to address the mission owner's high level question about the software..
3. **The composability of the software analysis approach or algorithm.** Rather than referring to the structure of the analyzer, this type of composability refers to how the algorithm approaches the problem of analyzing the program—does it attempt to reason about the entire program or does it reason about the program in smaller units, combining those small units of analysis into larger ones in a rigorous way. The latter is known as "Algebraic Program Analysis". Kincaid, Reps, and Cyphert put it this way: "A program analysis is compositional when the result of analyzing a composite program is a function of the results of analyzing its components."[23] Some of the analysis approaches contemplated in R1, R2, and R3 may leverage composable analyses and others may not.

The semantic composability challenge described in #2 above presents itself in many areas of this Roadmap. R1.4 will need to provide the mathematical foundations to address it, R2.2 will need to provide architectures leveraging these foundations, R2.3 will need to address it in the tool synthesis process, R3 and R4 will need to ensure that individual model components are consistent with these needs, and R5 will need to address these issues at the higher-level analysis campaign and orchestration level. R6 directly addresses the need to infer higher-level semantics from lower-level evidence.

Only the aggressive reuse of components at all levels can be scaled to the needs of Figure 2, and composability is a key to maximizing reuse of components.

# 4 Cross-Cutting Approaches

This roadmap organizes the software understanding needs of the nation into various research challenges. However, some important elements of the roadmap do not fit neatly into a single

[22] Robert Bartholet, *et al*, "In Search of the Philosopher's Stone: Simulation Composability Versus Component-Based Software Design," *Proceedings of the 2004 Fall Simulation Interoperability Workshop*, Orlando, FL, September 2004.
[23] Kincaid, Z., Reps, T., Cyphert, J. (2021). Algebraic Program Analysis. In: Silva, A., Leino, K.R.M. (eds) Computer Aided Verification. CAV 2021. Lecture Notes in Computer Science, vol 12759. Springer, Cham. https://doi.org/10.1007/978-3-030-81685-8_3.

research challenge, but instead are present in multiple places throughout the roadmap. Some of these elements are important enough to warrant special explanation as key, cross-cutting topics of the roadmap.

## 4.1 Cross-Cutting Approach, CC1: Dynamic and Static Analysis

*Dynamic analysis* is the process of running software under observation to learn something about it. Dynamic analysis exercises one (possibly lengthy) sequence of states of the software. Learning about additional states not exercised in the first run requires additional runs. In contrast, *static analysis* is the process of analyzing software without running it, drawing conclusions analytically by considering a set of states derived from the software artifact.

Each of these types has different strengths and weaknesses. Dynamic analysis can produce very detailed information (subject to dependencies on the experimental environment) for each run executed and is routinely used to find some unintended behavior; however, because most mission-relevant software contains more potential states than there are atoms in the universe, dynamic analysis is only capable of observing an infinitesimal subset of these. Dynamic analysis cannot explore states that the experimental setup does not exercise, and today's tools are unable to exercise all states of potential interest. Consequently, dynamic analysis alone cannot provide sufficient evidence for high assurance needs.

Static analysis, on the other hand, is capable of reasoning about many possible states simultaneously (in some cases, a very large number of states), but must make a tradeoff between precision and scalability—reasoning about more states requires eliding detail about those states, which can introduce both false positives and false negatives. Furthermore, static analysis of software removed from the context of the surrounding system typically requires modeling the missing elements of the system, which current tools typically cannot do adequately. Although static analysis can bring strong evidence to bear in high assurance cases, current technical limitations prevent it from being adequate today for reasoning about modern software systems.

Both R1 and R2 are needed to advance the state of the art in static, dynamic, and hybrid approaches to reasoning about software. R1.2 in particular is focused on strategies for making the precision/scalability tradeoff as well as identifying the information sources useful to guide it. The models of R3 and the approaches to automating the generation of those models of R4 will likely need different solutions for static and dynamic analysis.

## 4.2 Cross-Cutting Approach, CC2: Machine Learning, Artificial Intelligence, and Large Language Models

Machine Learning, Artificial Intelligence, and Large Language Models are technologies that show great promise for software and are likely to grow in importance in the coming decades. These technologies should be well understood both as a target of software understanding (because NS&CI software increasingly relies on these technologies) and as a method of software understanding (because these technologies can be leveraged for software analysis).

- **Machine Learning (ML)** refers to the use of any technique that allows an automated computer system to "learn" a function of interest based on experience (typically a large dataset of samples, with or without labels.) Machine Learning has been successfully used in a wide variety of applications to achieve near-human or even better-than-human level

performance, including chess, image recognition, driving, language processing, and some aspects of software understanding[24].

- **Artificial Intelligence (AI)** is a broader term than ML, encompassing *any* automated technique used to simulate, augment, or replace an intelligent actor making decisions. ML techniques are one example of AI[25].
- **Large Language Models (LLMs)** are a specific emerging technique within ML, which use a large computationally intensive model (typically a neural network) trained on a very large collection of data to understand and represent a target language (often human-generated text.) LLMs have been shown to perform phenomenally well at predicting and generating coherent text, for wide applications such as answering arbitrary user questions. The key advance behind LLMs is in the size and amount of computation utilized by the models and the enormous volume of data used to train the models. The same idea could serve as useful for understanding a large enough corpus of software as well[26].

For simplicity, in this document, we will loosely refer to any of the above as “ML and related techniques” or just “ML”.

There are multiple reasons why ML and related techniques are relevant to the subject of this roadmap. First, ML technologies are useful for analyzing software and gaining understanding. These techniques have proven to be valuable for a variety of software understanding problems and are likely underutilized as techniques today. For some software understanding tasks, these techniques may be the best or only ones available. This is the primary context in which this roadmap will discuss ML techniques.

Second, there are some emerging safety and security concerns that are specific to the use of ML technologies, such as model privacy and adversarial ML. While those issues are important, they are out of scope for this roadmap.

Lastly, sometimes ML tools or algorithms are used with NS&CI missions (e.g. ML technologies have been used in critical infrastructure control software). For these systems, it is worth noting that ML software is still *just software* for which mission owners still need to answer mission questions, just like other software. For example, one of the concerns of the field of AI Security is with the privacy of ML models; but if the underlying software has a traditional remote code execution vulnerability or an authentication backdoor, an adversary may be able to illicitly access the system and extract the raw information of the model directly from the system without having to use ML model attacks to infer it.

Because ML technologies can be used both to understand software as well as directly in NS&CI missions, ML technologies have an unusual dual role as both a target of software understanding and as an analysis technique for software understanding of other software systems.

---

[24] Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill.

[25] Based on definitions from both Congress and the White House, nearly all automated software understanding tools referenced in this document could also be considered a form of AI. In both 15 U.S.C. 9401(3) and EO 14110, “Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence” the definition of “AI” is: “a machine-based system that can, for a given set of human-defined objectives, make predictions, recommendations, or decisions influencing real or virtual environments. Artificial intelligence systems use machine- and human-based inputs to perceive real and virtual environments; abstract such perceptions into models through analysis in an automated manner; and use model inference to formulate options for information or action.”

[26] Jared Kaplan, *et al*, “Scaling Laws for Neural Language Models”, arXiv:2001.08361v1, 23 Jan 2020.

There are opportunities to apply ML technologies to novel problems in the software understanding space. This research roadmap does not include a specific top-level research thrust on ML because the authors believe that ML approaches will prove useful across many of the different research activities presented in this roadmap. It is important to note that, although the authors have done their best to call out where ML approaches may be applicable to specific software understanding problems, there are likely to be many situations where the authors have not foreseen where these techniques can be applied fruitfully. The reader is encouraged to think creatively about additional opportunities to apply ML techniques more broadly than just the ones listed in this roadmap.

There are some characteristics of software understanding problems that are well-suited to ML approaches, and some characteristics of software understanding problems that are not well-suited to ML approaches. Here is a brief a list of problem characteristics for which ML techniques should be considered particularly applicable, though this list is not meant by any means to be restrictive[27]:

- When a solution is expensive to find, but inexpensive to mathematically validate, ML can be used to make educated guesses for later validation. The external validation step is important for many use cases because answers provided by many ML tools can look plausible but be nonetheless incorrect depending on the particular training data and query.
- When precise mathematical answers are infeasible to compute, ML could provide probabilistically informed estimates which may be useful.
- When analysts want to aggregate multiple sources of imperfect, unaligned results (either through heuristics, or unsound analysis techniques, or human annotation) and they want to figure out which signals are the most important to synthesize an improved, final answer (see R6.3).
- When analysts lack complete context for the software under analysis and must infer models for the missing elements (see R4.3).
- When analysts have large quantities of data with labels relating to the function they want to learn.
- When analysts need to mediate machine-human interactions, such as responding to analyst's natural language questions or placing results into forms natural for humans to consume (see R2.4, R6.2, and R7).
- When analysts need to configure analysis parameters, ML can be helpful in searching for optimal settings to answer particular questions (see R2).
- When analysts want to recognize or leverage a set of fuzzy patterns, such as in semantic inferencing (see R6).

Additionally, there are several ways in which traditional software analysis tools and ML tools could be of mutual benefit:

- The software understanding components discussed in this roadmap could provide a novel and much richer features vector for ML training than is available today.
- Software understanding often involves too many options to explore; ML techniques could be used to prioritize options to improve efficiency (see R2.5).

[27] For more insight into how to effectively apply ML techniques to software problems, see Pedro Domingos, "A few useful things to know about machine learning," October 1, 2012, Communications of the ACM, Volume 55, Issue 10, pages 78–87. https://doi.org/10.1145/2347736.2347755.

- Formal approaches to software understanding often become non-performant when complete data is not available. By training on situations in which complete information is available, ML techniques may be able to make mission-actionable estimates in cases in which full information is not available and the formal tools would provide no actionable results.
- The roadmap below discusses creating numerous small components that can be reused through configuration and composition to make a given analysis tool. Selecting and configuring a large number of small, configurable components to achieve an end is a tremendous optimization challenge, which ML approaches may be well suited to tackling (see R5.4).
- The roadmap below further considers that a successful analysis may involve decomposing high-level mission questions into smaller, more focused questions about the details of software state, executing an iterative sequence of analyses to tune precision to an optimal balance of scalability and precision, and then composing the resulting data into an actionable answer (see Figure 3 and R5 for more discussion). ML techniques may be suited to orchestrating such an analysis campaign.

## 4.3 Cross-Cutting Approach, CC3: Human Factors and Human Studies

The goal of this roadmap is to guide the research needed to create a vastly improved, automated national software understanding capability. However, the current reality is that software understanding questions for mission problems are answered largely through the manual effort of human reverse engineers and analysts. An automated software understanding capability requires a thorough understanding of what human reverse engineers and analysts do today to arrive at answers to mission questions.

Automation of the software understanding processes is not an all-or-nothing endeavor. There is a graduated spectrum from fully manual to fully automated. When anyone takes a process that is largely manual and create or improve tools that help with tasks within that process, they are moving incrementally towards the automated end of the spectrum. In general, one should not expect to move from the manual end of the spectrum (which is the state of the practice now with most software understanding questions) all the way to the fully automated end of the spectrum with a single tool, approach, or research advance.

Automating the work that is done manually today is a difficult task that is easy to underestimate. Although some preliminary work has been done[28,29,30], this area has not yet been sufficiently studied. An effective automated software understanding capability requires a sufficiently deep understanding of the manual process to prioritize which activities are most appropriate for early automation, guide initial efforts in automation, and to inform smooth integration of that automation with human analysts' workflows. And finally, an automated software understanding capability requires a deep understanding of the manual process to improve efficiency of tasks that are not currently automatable. The authors expect that there will be a set of incremental advances that

[28] Nyre-Yu, M., Butler, K. M., & Bolstad, C. (2022). A Task Analysis of Static Binary Reverse Engineering for Security. Hawai'i International Conference for System Sciences, Honolulu, HI.
[29] Matzen, Laura E., Leger, Michelle A., & Reedy, Geoffrey (2021). Effects of Precise and Imprecise Value-Set Analysis (VSA) Information on Manual Code Analysis. Workshop on Binary Analysis Research (BAR) 2021, Virtual.
[30] Butler, K., Leger, M., Bueno, D., Cuellar, C., Haass, M. J., Loffredo, T., Reedy, G., & Tuminaro, J. (2019). Creating a User-Centric Data Flow Visualization. Human-Computer Interaction International 2019, Orlando, FL.

move the national software understanding capability closer and closer towards automation until the capability reaches a point that is satisfactory for mission needs. To this end, engaging in Human Factors analysis and Human Studies to understand how to interface between human analysts and semi-automated tools is vital.

Regarding the development and use of automated software understanding capabilities for NS&CI missions, there are at least four different groups of individuals with different activities that human factors investigations will need to consider:

1. Manual reverse engineers who today perform limited software understanding tasks manually (as mentioned above and addressed in R6.2), examining software artifacts in detail to gather technical evidence related to the mission question.
2. Mission owners, posing questions, and interacting with the evidence package roll-up (see R7).
3. Users of analysis tools in operational settings, such as malware analysts at security operation centers who today use existing automated tools to gather information about software for mission questions. Since moving from manual to fully automated workflows cannot happen instantly, these human analysts will continue to be vital elements in increasingly automated workflows for many years.
4. Analysis tool developers, who need to inspect and debug analysis tools and orchestrate analysis campaign algorithms, which involves analyzing the huge volumes of data being processed by the tools as well as understanding the information needs of analysts.

A national software understanding effort needs to perform human studies on each of these groups to understand the types of questions they need to pose to the analysis system, and natural methods of expression of this information. That software understanding effort must translate from these human-centric representations to precise, well-defined machine-readable representations that can be used to reason hierarchically about questions and evidence, synthesize appropriate analysis tools, and orchestrate those into an analysis campaign (see R7, R2, and R5.4). LLM approaches from CC2 may apply to many of the human interface needs described below.

For reverse engineers, human factors studies are needed to identify the parts of the manual analysis process that are good candidates for automation, and to translate between the input/output of specialized tools and the analysts to facilitate their analysis activities.

For mission owners, human studies are needed to guide the output of automated software understanding tools and systems, including what information is most useful in an evidence package to enable high-level decision makers to make risk-mitigation and -acceptance decisions. The data must succinctly summarize results while reflecting the completeness, confidence, uncertainty, etc. of the analysis process.

For analysts using software understanding tools in operational settings, human factors are needed as well. As mentioned above, for the foreseeable future, analysis campaigns will typically require the integration of human analysts interacting with tools for key steps. Thus, tools and human analysts will need to work seamlessly in concert. Software understanding tools operate over nigh-infinite state spaces with huge volumes of data, which analysts cannot possibly hope to sift through manually. This data is often replete with false positive and false negatives depending on the algorithms used and the configuration parameters. Analysts must be able to request a variety of types of summarizations of the large state spaces of data, search in natural ways for detailed

information, trace provenance of select data and review sources for assessing confidence (see CC4). Results of these queries should be presented in modalities most interpretable and useful to the human analysts, enabling zero-friction, interactive inquiry.

For analysis tool developers, many of the needs of the analysis users described in the previous paragraph apply. Developers need to sift through analysis data to diagnose problems and validate answers, requesting summarizations of large state spaces, searching in natural ways for detailed information, tracing data provenance, and more. Developers have the additional need of performing those same functions on internal, intermediate data of any given analysis tool during the development process. To the extent that various tools in an ecosystem (see FA1) all need to accommodate similar debugging needs, standard debugging strategies and approaches informed by human studies may be valuable.

At a minimum, the following areas of study are needed across many of the research thrusts in this roadmap:

- Studies of human analysts and tool developers to gain insight into the current manual processes and reasoning that inform their work, including what and how information is used to assess software.
- Studies of mission question owners and decision makers to understand what and how information is needed to make decisions.
- Tool interface studies to select appropriate evidence and improve how it is presented to human decision makers with a particular emphasis on reverse engineers doing software understanding. Human studies are needed to identify strategies to accelerate (1) the identification of relevant data and (2) the interpretation of that data.

## 4.4 Cross-Cutting Approach, CC4: Design for Introspection and Debugging

In other fields, the concepts of *design for manufacturability* and *design for test* are well-understood and embraced. Similarly, in the field of software understanding tools, maintaining healthy progress on the roadmap explored here will require *design for introspection* or *design for debugging*. The volume of data involved in software understanding is staggering. Once an answer to a given mission question is generated, a natural corollary question is one of explainability and introspection: *why did the system generate this answer*? *What data from the binary led to this conclusion?*

During research and development phases before a tool is complete an answer can be generated, when something goes wrong (and it will), it is a monumental challenge to sift through the data to understand and address the problem. Developers must have rich, useful, natural ways to rapidly determine what is wrong in the enormous state spaces involved, to understand why it is wrong, to identify root causes, to manually engage in “what if” experiments by overriding key intermediate information, and to design improved algorithms to correct the shortcomings.

Given the challenges, calls for research proposals aligned with this roadmap should be willing to entertain explicit research activities to improve introspection, explainability, and debuggability. Developers will need special tools designed to aid them in debugging. This should involve human factors studies to understand the debugging needs of the developers and to identify human-centric ways of displaying and searching the vast amounts of data. This should involve ML techniques to aid the developer in sifting through the data as well as large language models for interacting with the human developer.

The details will vary depending on the specifics of the research thrust being supported, but here are some general strategies for pursuing introspection, explainability, and debugging:

- Perform human studies on the information needs of the developer/analyst.
- Develop algorithms (whether formal or ML based) to enable rapid and expressive querying of the system data by the developer/analyst.
- Innovate summarization approaches (whether formal or ML based) to reduce the volume of data in concise yet semantically meaningful ways.
- Develop approaches for translating, meaningfully summarizing, and presenting complex analysis state of programs for the human analyst, driven by the cognitive processes and knowledge requirements of the human analysis.
- Investigate approaches for fusing data from disparate sources to enable cross-domain introspection, explainability, and debugging.
- Develop approaches to facilitate developer hypothesis exploration by speculatively changing system state, running the system for a time, and inspecting the resulting changes to the analysis progression.

# 5 Research, Development, and Engineering Roadmap

This section presents a first version of a technical roadmap to address national needs for software understanding in national security and critical infrastructure mission spaces. Ideally, any owner over a national security or critical infrastructure mission would be able to routinely pose and receive answers to any mission question regarding any software upon which their mission depends. Such answers could be received in mission-relevant timeframes at practical expense. This roadmap focuses on the long-term research and development needed to inform the creation of such aspirational software understanding capabilities. The scope and size of the overall national need requires an enduring, national-scale research program, perhaps similar to the national efforts in combatting cancer[31]. This roadmap is meant to inform the first 5-10 years of that program.

The authors fully expect that this roadmap will need to change in the future—perhaps radically—as more is learned. The authors consider this Roadmap to be merely the first version of what will certainly be many versions. At a high level, this first version of the roadmap adopts a particular structure and emphasis for approaching the research which future revisions may improve upon or abandon altogether in favor of something better.

This roadmap is expansive, including things that *may* need to be done, not just those things which *must* be done. No one knows enough yet to confidently produce a roadmap to the capabilities needed, without some amount of speculation. Consequently, an essential aspect of early research on this roadmap will involve experimenting with ideas that may not succeed. This approach is required to enable researchers to acquire the necessary knowledge to guide the subsequent iterations of this roadmap. To support this, early work on the roadmap should take the form of experiments, not to produce a mission-impactful outcome directly, but to learn more about the essence and nature of the problem and to more deeply characterize the presently available techniques and approaches.

[31] Otis Brawley and Paul Goldberg, "The 50 Year's War: The History and Outcomes of the National Cancer Act of 1971," Cancer, Volume 127, Issue 24, 15 December 2021. https://doi.org/10.1002/cncr.34040

The research and development (R&D) and engineering needs outlined in this document could be addressed using a wide range of approaches. Some of these approaches are well-documented in academic literature and are nearly ready for implementation, while others involve problem areas where no viable approach is currently known. In certain cases, this roadmap could be pursued by adopting or adapting existing methods, while in others, progress will require improving or refining work that has been pioneered in academic contexts but requires further research. There will also be instances where the focus is on studying problems or innovating new ideas where reasonable approaches have not yet been identified. All of these degrees of research are crucial for establishing a robust Software Understanding capability. Furthermore, in many emerging fields of study mentioned in this roadmap, trial and error is a key method of progress—it is important to anticipate that the difficulty of the challenges faced means that research progress depends upon learning from both failures and successes. It is a mistake to prematurely strive for a comprehensive solution.

Due to the fundamental characteristics of software discussed in "The National Need for Software Understanding"[1], Section 4, even in an ideal world, not all mission questions or software under test can be analyzed in a fully automated fashion, with perfect precision, guaranteed to always work. In important cases, software reverse engineers would still be needed to configure tools, monitor and direct analytics, or complete challenging analyses. Research conducted pursuant to this roadmap should be about *learning and progress*, not *production or perfection*.

Though this roadmap primarily emphasizes R&D activities, it is important to note that there is also an analogous need for engineering work to put R&D advances into practice. Research work and engineering work should generally be done as separate activities, because they have sometimes conflicting goals; the goal of research work is to explore, learn, understand, document, and/or demonstrate something for the future to build upon, while the goal of engineering work is to create something useful and impactful to mission based on what is understood at the time. Research efforts and engineering efforts in software understanding complement each other, and both are important to create the necessary capability. At the same time, research must be informed by mission needs to avoid pursuing avenues of discovery with no plausible path toward impacting the national needs.

The technical thrusts of this roadmap can be visualized (in a somewhat oversimplified fashion) in Figure 3. This figure represents a software understanding *campaign*, in which a high-level mission question is decomposed into specific, low-level questions about the software's state that prescribe the execution of a series of analysis tools, producing output which is rolled up into an evidence package to present to the mission owner originally posing the mission question. Among other things, the analysis tools themselves are assessed using a suite of benchmarks. The yellow "R" numbers below each blue box represent the research thrust(s) of this roadmap which address the needs of that stage in this process. The hierarchical breakdown of the mission question and roll-up of evidence are discussed in R7, the benchmarks in R8, while the research thrusts involved in creating the analysis tools themselves are overviewed in the next paragraph. For a detailed illustration of Figure 3, R5 presents an extended, hypothetical example.

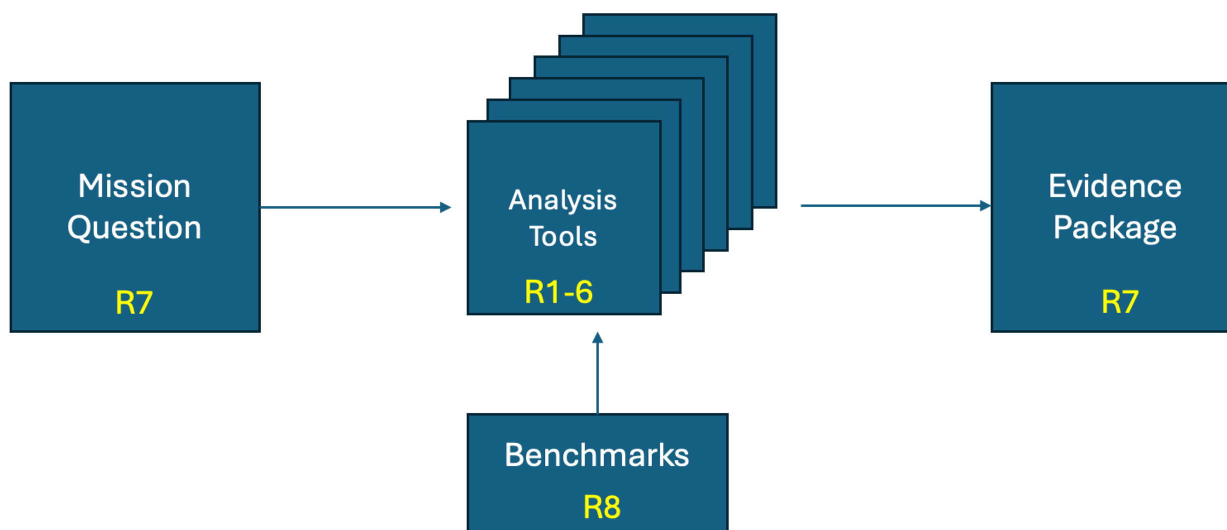


*Figure 3: Notional overview of a software understanding campaign.*
*This diagram illustrates how the various research thrusts of this roadmap fit into an analysis campaign.*

Figure 4 depicts the relationship of the research thrusts of this roadmap related to constructing a single analysis tool. This roadmap starts with the foundations of reasoning about software (R1), proceeds to the architectures for individual analysis tools (R2) and the software models they leverage (R3) and how to generate them (R4), and then considers the ways that the ecosystems of software analysis tools will need to work in concert (R5). A vital function of the elements that comprise the analysis tool is to infer higher level semantics from lower-level information (R6), similar to making high-level conclusions about the picture of a puzzle by examining the individual puzzle pieces, or inferring conclusions about a crime by looking at a set of low-level forensic evidence.

The presentation structure of this roadmap is a series of research thrusts and challenges, each containing multiple sub-areas, each of which in turn lists specific research needs, like so:

R&D Thrust, R[N]: Research Thrust Name

R[N.M]: Sub-Area Within Thrust

- Bulleted list of the specific research project/program needed.

These research thrusts are not independent, with overlap, dependencies, and synergies among many of these research needs. The authors will endeavor to call these out as they arise.

For a large area of research like software understanding, it can be difficult to know whether sufficient progress is being made over time, or how much apparent progress there is. The authors aim to capture these vital needs under R8, Datasets, Benchmarks, and Ground Truth. Improving the metrics for successful software understanding by developing datasets and benchmarks can provide the nation with objective ways to measure progress over time. By including a range of problem difficulties, including both real and synthetic samples, these datasets and benchmarks can help software understanding research first learn to crawl, then learn to walk, then learn to run, before it becomes a robust capability.

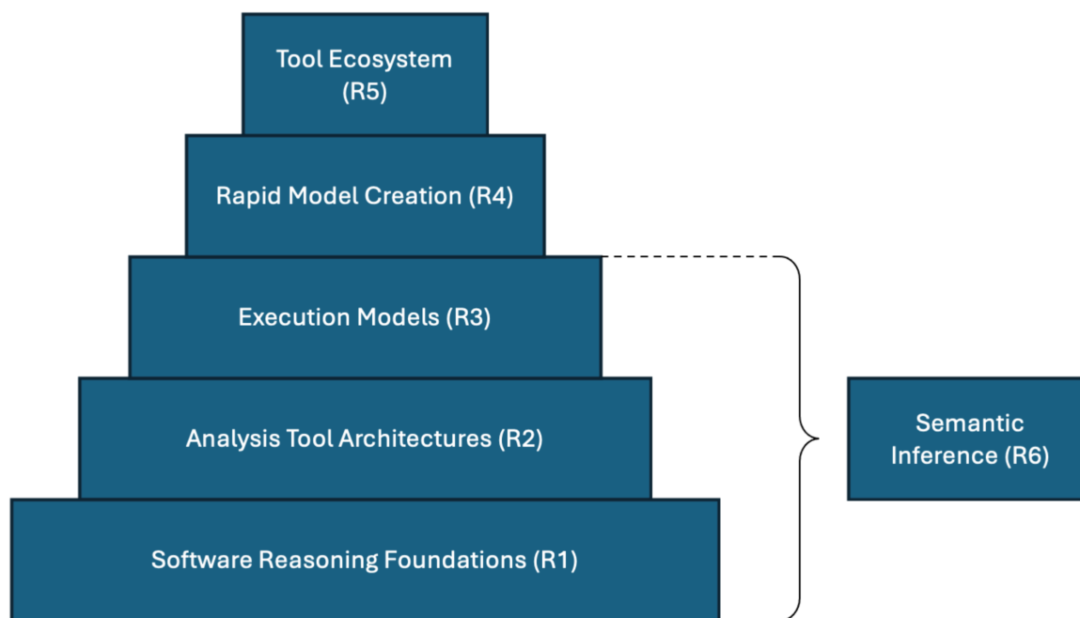


*Figure 4: Conceptual relationship of R1-R5 in this roadmap.*
*R6 calls out additional challenges that span more than one of the R1-5 thrusts.*

## 5.1 R&D Challenge, R1: Formal Foundations for Software Reasoning

To trust software in national security and critical infrastructure mission spaces, mission owners must know with confidence in advance what behaviors that software could exhibit. The highest confidence approaches to understanding what behaviors software could exhibit require modeling the execution of the software in its execution environment, enabling analytical methods to explore the software's possible behavior prior to putting it into use. Such analyses could be used to assess the inherent risk to mission of using the software, guiding mitigations, and enabling risk acceptance decisions to be based on technical evidence derived from root causes.

The vast majority of software being produced today is not designed to facilitate such modeling and analysis. While some aspects of software are simple to model, many are challenging. What modeling options are needed, how they should be selected to model a given software artifact, and how to compose those models together to make an analysis tool, are all underexplored areas.

The use of math to model systems to predict behavior is the foundation of modern engineering and many branches of physical science. Software is different from physical systems such as a ship or a bridge and requires novel approaches adapted to its peculiar characteristics, but the foundational approach of mathematical modeling applies similarly to software as it does to those other disciplines. In its essence, software is a mathematically inspired construct, with CPU instructions manipulating numbers in precise ways to accomplish the software's overall functionality. There are many positive results from academia, industry, and open-source code in which software has been modeled to reason about its behavior in simple cases. Although perfection may not be possible in all cases,[32] there is tremendous room for mission-impactful progress. Concerted effort is needed to

[32] For a discussion on the Halting Problem and Rice's Theorem and how those relate to the technical challenges and limitations of software understanding, see Section 4.3.4 in D. Ghormley, T. Amon, C. Harrison, and T. Loffredo, "The National Need for Software Understanding", Sandia National Laboratories, Jan 17, 2024.

mature these approaches, to scale them to mission-relevant problems, and to identify key gaps where innovation is required.

This research thrust focuses on mathematical and scientific foundations necessary to reason about software—the set of logics and their characteristics needed to underpin any software analysis tool. Later research thrusts will focus on the specific applications of modeling various elements of software systems (R3), challenges in generating those models (R4) and how to stitch those models into analysis tools (R2). The models and tool architectures will need to be designed within some mathematical logic system, which is the focus of this section.

### 5.1.1 R1.1 Reasoning logics

The best techniques today for reasoning with rigor about software behavior require translating the software's semantics into mathematically based, logical systems and then reasoning within those logical systems to answer particular questions about the software's behavior. Different questions may require different logics or different translations. For example, questions about temporal characteristics may require temporal logics, whereas questions about privacy may involve separation logics.

Over the course of many years, the academic community has developed and studied an extensive set of semantics, logics, and algorithms related to analyzing software, including denotational, operational, and axiomatic semantics; big and small step semantics; collecting and temporal semantics; separation logic; abstract interpretation; model checking; constraint logics; and many more.

An open question is what fraction of the overall national software understanding need is addressed by these logic systems and what residual gap remains? Logical systems such as these need to be studied with respect to the high-level mission questions (discussed in BR1 and depicted in Figure 2), with respect to composing them, and with respect to generating evidence about the behavior of programs.

To answer a high-level mission question, a software understanding analysis tool may need to leverage numerous semantic and logic systems, interoperating in concert. In many cases, existing semantic and logic systems will need to be extended in order to achieve the characteristics addressed in R1.2-R1.4. Novel semantic and logical systems will likely need to be innovated to fill critical gaps. Finally, since different MQs and PUTs may require different analyses, analysts need analyzer synthesis approaches which can map the various semantic and logic systems onto the needs of a given analysis task.

To answer high-level mission questions about software, research, development, and engineering are needed in the following areas:

- Study a broad, representative variety of high-level mission questions to determine what types and combinations of reasoning logics are needed for various aspects of different mission questions (this relates to mission questions decomposition discussed in R7).
- Survey existing reasoning logics to assess the maturity and adequacy of existing systems relative to the needs identified in the preceding bullet.
- Develop proof-of-concept prototype analyzers for mission questions requiring a variety of reasoning logics to study the challenges of combining those logics.
- Develop approaches that can marshal/synthesize multiple semantic and logic systems together, interoperating in concert in an aggregate analysis.

- Assess currently available semantic and logic systems to identify key gaps relative to the national software understanding needs.
- Extend existing semantic and logic systems as needed to exhibit the characteristics discussed in R1.2-R1.4.
- Innovate novel semantic and logic systems needed to fill gaps in the current ability to reason about software.
- Develop approaches to map software analysis needs to solution candidates, enabling analysis synthesis strategies to identify options when synthesizing a complex analysis.

### 5.1.2 R1.2 Approaches for Guiding Precision vs. Abstraction Tradeoffs

Modern real-world programs have more potential states than there are atoms in the universe—in such cases, it is impossible to fully explore all possible states. To achieve the rigor and scalability necessary for mission-relevant systems and analysis timeframes, software behavior details irrelevant to the analysis question at hand must be aggressively elided, while sufficient relevant detail is retained with enough precision to provide the technical basis for the needed answer. Knowing what information to elide and what information to retain, and with what precision, is a nontrivial challenge. Because of the exponential nature of software, improvements in this space may *exponentially reduce* the compute and memory requirements for a given analysis. That is, to the extent successful, this research thrust could transform analyses from impossible to practical.

Research, development, and engineering are needed in the following areas:

- Study a broad variety of examples of mission questions and software samples from the perspective of the precision and scalability needed for different stages in the mission question decomposition and evidence composition (see R6).
- Extend existing or develop novel semantics and logics to adaptively retain or abstract detail based on an external analysis policy.
- Develop methodologies (whether iterative or *a priori*, whether formal, heuristic, or AI/ML-informed) for guiding analysis policy for the retention or abstraction of analysis detail based on the needs of the analysis question, the software under test, and available resources. (The authors note that the concept of *abstraction refinement* is one potentially valuable approach to leverage.)
- Develop hierarchical reasoning strategies which can retain or elide precision as analysis results are "rolled up" to produce an evidence case, to ensure that explainability and data provenance can be maintained (this relates to R6).
- Study human analysts and innovate strategies for integrating them with automation pipelines to guide or override analysis precision vs. abstraction tradeoffs based on analyst needs.
- Explore ML approaches to heuristically guide precision vs. abstraction tradeoffs based on the mission question, the program under analysis, analysis resources, timeliness requirements, and mission owner confidence needs.

### 5.1.3 R1.3 Harnessing Modern and Emerging Hardware Architectures

Modern computing infrastructure innovations continue to offer higher performance through more parallel and specialized hardware architectures, including traditional high-performance architectures, multi-core systems, vector-based hardware, graphical processing units (GPUs), Tensor Processing Units (TPUs), and more. As an inspirational example, AI/ML algorithms have experienced

enormous improvements in capability and performance by developing algorithms which can leverage such architectures. In contrast, software analysis tools have yet to achieve similar benefits.

Furthermore, hardware architectures have various operating regimes whose performance can differ by orders of magnitude (e.g. hitting a cache vs. reading from a remote memory in a NUMA system). Although programs are not required to be aware of such modes for correctness, tuning an algorithm to take maximal advantage of such performance benefits has the potential to improve analysis performance by orders of magnitude.[33]

There are multiple aspects of scalability to be explored as part of this roadmap. This research thrust focuses on performance scalability related to harnessing novel and emerging computing architecture advances. Whereas the research thrust in R1.2 seeks to lower the computational and data burden of software analysis, this research thrust seeks to bring more and novel computational power to bear on the problem of analyzing software—the two thrusts are thus complementary.

Research, development, and engineering are needed in the following areas:

- Study the applicability of recent and emerging hardware architecture innovations to software analysis tasks.
- Develop novel software analysis algorithms designed to take advantage of recent and emerging non-traditional single-machine architectures to dramatically improve performance.
- Develop cluster-centric software analysis architectures, capable of taking advantage of the distributed computation and data stores of modern elastic cloud infrastructures.
- Adapt existing analysis algorithms to be tunable to take maximal advantage of highly performant operating regimes of modern architectures.

### 5.1.4 R1.4 Compositional Reasoning

The scope of the NS&CI mission needs in analyzing software are vast. As discussed in FA1, analysis tools are a function of the mission question and program being evaluated, among other things. Creating a whole-cloth, bespoke analysis solution for the combination of each mission question and software system of national relevance, without extensive re-use of analysis elements, would be catastrophically expensive. Practical considerations demand aggressively pursuing opportunities to maximizing economies of scale through reuse. This requires identifying approaches that maximize reuse of components across different mission questions, across different software systems, across mission areas, and across government departments and agencies. Technically, this requires synthesizing software analysis tools from reusable, composable components to the maximum extent feasible. Based on the opinions expressed by technical SMEs participating in SUNS 2023, a positive return on investment (ROI) is expected.[1]

This need for compositional reasoning is not isolated to research thrust R1, but also arises in the analysis architectures of R2, the models of R3 and R4, the tool ecosystem of R5, and the hierarchical reasoning of R7. Within R1, the semantics and logics from R1.1 must be extended to work compositionally, the approaches for guiding precision vs. abstraction tradeoffs in R1.2 must

---

[33] Oded Green, Robert McColl, and David A. Bader. 2012. GPU merge path: a GPU merging algorithm. In Proceedings of the 26th ACM international conference on Supercomputing (ICS '12). Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/2304576.2304621.

accommodate composition, and the solutions to harnessing modern architectures should be pursued to maximize composition to the maximum extent possible.

There are several existing compositional techniques for software analysis which have already been explored in industry and academia, and even more software analysis constructs that may be leveraged in novel compositional ways. What is not yet sufficiently understood is the overall adequacy of these techniques collectively relative to the national needs in real mission systems in national security and critical infrastructure. It is to be expected that much more work needs to be done to assess existing options, identify gaps, and mature promising areas.

Beyond the mechanics of composability, the semantic composability problem must be addressed for the logic systems employed in software understanding. The semantic composability problem focuses on ensuring that the aggregate composition of components has the intended behavior, achieves the desired goal, and exhibits the desired emergent properties. (See R2 for more discussion.)

Research, development, and engineering are needed in the following areas:

- Study existing academic and industrial approaches to compositional software analysis, creating metrics for assessment to identify gaps and promising areas for further investment.
- Identify and explore options for composing the semantic and logic systems of R1.1, advancing the theory of these systems as necessary to enable composability.
- Innovate solutions to the semantic composability problem of composing these logic systems (see R2).
- Research novel approaches to compositional reasoning systems that can effectively separate the reasoning logic into distinct, reusable, interchangeable components.
- Evaluate the effectiveness of these compositional analysis techniques on realistic problems, to find the most promising compositional reasoning systems.
- Develop a library of options to be leveraged by promising approaches capable of compositional reasoning. For example, create a library of abstract domains for an abstraction interpretation system that can be composed using a *reduced product* approach.

#### 5.1.5 R1.5 Machine Learning (ML) Approaches to Reasoning Systems

In mission-relevant systems, formal reasoning can meet with barriers or constraints that cause the formal reasoning to break down, making formal techniques alone insufficient to meet the full scope of the national needs. While some obstacles can be overcome through novel logics, abstractions, or modeling approaches, others cannot. ML has the potential to achieve adequate software understanding in these cases, even though those solutions may often lack the mathematical guarantees of sound and complete formal reasoning. Furthermore, using ML techniques in concert with formal reasoning could potentially provide better analysis results than either technique could alone. Additionally, novel formal techniques may take significant investment and time to develop, whereas ML may be able to provide probabilistic answers on a more aggressive timeline than more formal approaches. Finally, ML can provide results based on practical data analysis of real-world problems that are too complex to successfully reason about formally.

Research, development, and engineering are needed in the following areas:

- Study the gap of what technical obstacles are unsolvable today using current formal reasoning systems and evaluate which may be amenable to ML.

- Identify opportunities and establish enduring efforts to gather large datasets of mission-informed software samples, mission questions, manual reverse engineering efforts, and more, ensuring that the resulting datasets are tiered by sensitivity and available as appropriate to researchers both within and outside the government, both domestic and international.
- In cases in the above datasets in which mission sensitivities prevent sharing, develop techniques to sanitize data to enable broader sharing while preserving sufficiently useful information to provide value to researchers.
- Study extensions of Uncertainty Quantification (UQ) and explainability targeted to these combined problems.
- Study existing techniques for combining information from varying levels of confidence, including formal reasoning, ML approaches, and human-sourced results.
- Investigate novel techniques to combine information from varying levels of confidence, including formal reasoning, ML approaches, and human-sourced results.

## 5.2 R&D Challenge, R2: Analysis Architectures and Automated Tool Synthesis

Mission success commensurate with the national need depicted in Figure 2 requires innovating analysis architectures that can scale up by numerous orders of magnitude.

As discussed in FA1, an analysis tool is a function of the program under evaluation, the question to be answered, and the resources available. The analysis tool needed to analyze a x86-32 binary vs. Java bytecode are quite different. The analysis tools needed to analyze (1) malware for its command-and-control protocol, (2) maritime ports crane software for hidden triggers, (3) security software for authentication bypasses, and (4) all software for exploitable vulnerabilities, are all very different. The analysis tool which must produce an answer fast enough to run between the time a user clicks on an application and when it runs is very different than the tool that will run on cloud infrastructure for a month to analyze NS&CI software with maximal rigor.

Mission success requires analysis architectures that can scale across needs of national security and critical infrastructure systems, including the wide breadth of software, the wide range of mission questions, and the significant resources needed for the nation's most highly consequential strategic systems. To maximize reuse of components, any given analysis tool will be a synthesis of numerous composable components (see FA2), whether the logic building blocks of R1 or the execution models of R3. Furthermore, this collection of components will need to perform the task of inferring higher-level information from low-level details in a bottom-up analysis (see R6).

In this process of selecting the building blocks to synthesize, there are numerous decisions to be made driven by the purpose of the analysis tool, including what technical evidence regarding the program needs to be tracked, how it will be need to be presented upon completion (which informs what details will need to be retained during analysis), what constructs in the target program will need to be modeled and with what precision, which search strategies to use to guide the analysis, and many more. Each decision to be made requires an array of available options, metrics for assessing those options, and approaches for assessing the options according to those metrics.

Furthermore, in the process of tool synthesis from building blocks, considerable additional work is required beyond what the low-level building blocks provide. Today, this work is almost entirely manual. To produce timely answers at the pace of mission in a fast-paced cyber world, this process of tool synthesis will need to be heavily automated, involving the human analysis tool writer in limited, key points of the process.

No single analysis architecture can satisfy the variety of needs above. Innovation is needed to accommodate the wide variety of use cases needed across various missions, to scale existing approaches to the size and complexity of mission-relevant software, and to get answers in mission-relevant timescales.

Although the challenges are daunting, there are decades of academic foundational work in software analysis architectures that can be leveraged as starting points. As just one example, the seminal paper in Abstract Interpretation was published in 1971,[34] with over 1000 academic paper published in the area in the years since.[35]

### 5.2.1 R2.1 Analysis Structures and Techniques

Academic efforts have explored a variety of analysis formulations over the years, including monadic abstract interpreters, work list approaches, fixed-point analyses, dataflow methods, formal inference rules, symbolic execution, and more. Different analysis techniques and structures have different characteristics, with advantages and disadvantages that can be matched with the varying needs of different kinds of mission problems; no single approach is appropriate for all analysis tasks. But these advantages and disadvantages have not been studied in any rigorous way or synthesized into a standard practice for effective analysis creation. Furthermore, these largely academic efforts are not mature enough to be applied to mission-scale problems today.

Academics have also innovated novel languages to support software understanding. While several interesting options are now available, it is as yet unclear which of these is optimal for which characteristics of the software application under analysis or the mission question. It is also unclear whether the software understanding community at large has yet discovered all the options that will be needed to be successful at meeting the national needs in software understanding. This line of research should be pursued to identify additional options, learn the conditions under which different options are optimal, and innovate approaches to blend different options together into a single analysis when appropriate.

Research, development, and engineering are needed in the following areas:

- Study existing software analysis structures relative to the needs of real-world mission systems in national security and critical infrastructure, characterizing available options to capture their relative advantages and disadvantages, identifying existing work that is relevant and adequate, existing work in need of maturation investments, and core gaps in need of initial investment and innovation.
- Advance the theory of software analysis machines to broaden the set of analysis architecture options for scaling to real-world national needs.
- Create processes for selecting optimal analysis architectures and components for a given analysis question and software under analysis.
- Advance the state of automating the synthesis of a given analysis tool, to dramatically reduce the manual effort required to create and validate a new tool.

---

[34] Cousot, Patrick and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (1977).
[35] According to a search on https://dblp.org of the term "abstract interpretation", performed on June 28, 2024.

- Advance research in promising abstract interpretation methods: Abstracting Definitional Machines[36], Abstracting Abstract Machines[37], and Staged Abstract Interpreters[38]. These efforts represent research on the way that abstract interpretation machines are structured in order to achieve different characteristics/benefits of the machine. This line of research should be continued, researching interpreter structures that can leverage the innovations of R1 and achieve many of the other characteristics in R2 and R3.
- Develop a standard of practice for deciding how to structure an analysis based on the needs of the mission. This includes what analysis approach to adopt, what analysis architecture is most appropriate, what analysis domains are necessary, and what precision strategies are necessary for the novel analysis.
- Formulate analysis architecture designs able to maximally harness the of hardware architectures from R1.3, into a single analysis tool.

### 5.2.2 R2.2 Composable Analysis Architectures

Although several existing analysis tools today reflect some elements of composability (in the sense of FA2, facet #2), or at least modularity, none have been designed with the full scale of the national problem in view. In many cases, existing analysis tools have hard-coded either the mission question, details of the software under evaluation, assumptions about available resources, or are just written in a one-off, monolithic, not-reusable fashion. Researching effective ways to develop reusable, reconfigurable, composable analyses is key to reversing this trend.

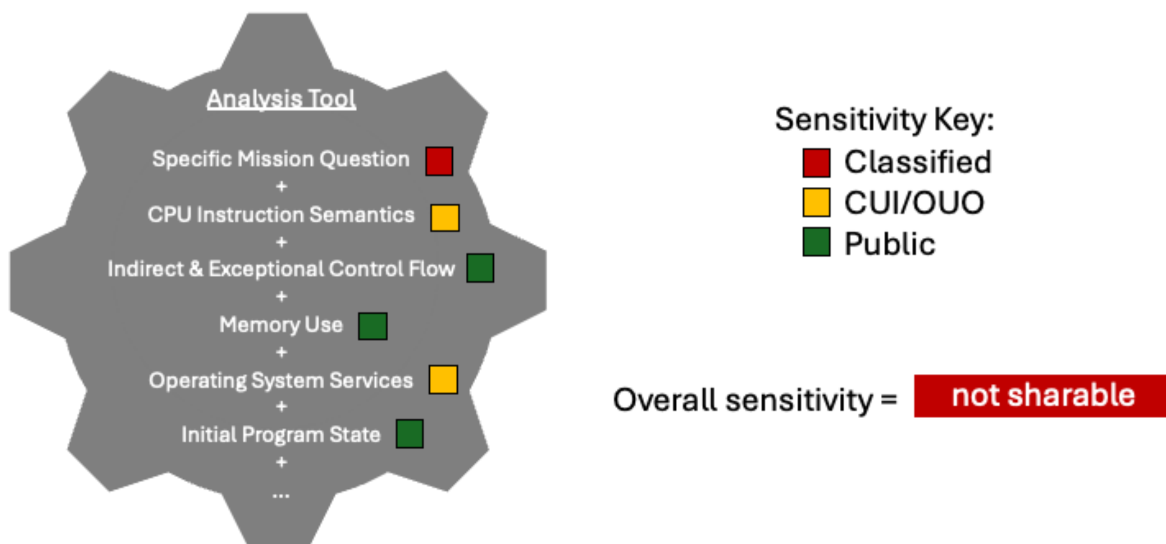


*Figure 5: Notional depiction of a monolithic tool.*

[36] Darais, David, et al. "Abstracting Definitional Interpreters". In *Proceedings of the ACM on Programming Languages 1.ICFP (2017): 1-25*. https://arxiv.org/abs/1707.04755.
[37] Van Horn, David, and Might, Matthew. "Abstracting abstract machines." In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming, 2010*. https://arxiv.org/abs/1007.4446.
[38] Wei, Guannan, et al. "Staged abstract interpreters." In *Proceedings of the ACM on Programming Languages, Vol 3: 1-32 (2019).* https://dl.acm.org/doi/10.1145/3360552.

*The grey gear represents a monolithic tool developed with elements of varying sensitivity. The lack of componentization means the overall tool is as restricted as the most sensitive element.*

In addition to the traditional benefits that sharing reusable components across the USG would provide, in NS&CI mission spaces there is another benefit—addressing sensitivity and sharing concerns. Today, tools that are built monolithically have sensitive elements mixed with commonly sharable elements (Figure 5), resulting in an overall inability to share the tool. If, instead, analysis tools were constructed largely from reusable components, then components could have individual sharing sensitivities. Consider a hypothetical sequence of stages of sensitive information in these analysis tools:

- Stage 0: Fundamental Research – Public
- Stage 1: Protected Advancements – CUI/OUO
- Stage 2: Mission-Specific Elements – distribution limited by classification, need-to-know, suitability, etc. as needed

Operating under such a system, components categorized as Stage 0 could be shared very broadly, receiving full benefit from collaboration with academic, commercial, and other public researchers, including foreign researchers; components categorized as Stage 1 could be shared somewhat broadly to government-affiliated researchers; only components categorized as Stage 2 (a very small set, hopefully) would require highly restrictive controls. Most of the work currently done for software understanding in the USG is in Stages 0 or 1, but must be handled as with Stage 2 restrictions, because of the lack of componentization. This results in duplicated effort, waste, and enormous missed opportunities.

The composability contemplated here is that of analysis tool components (FA2, #2), as opposed to the composability contemplated in R1.4, which is that of foundational logic systems (FA2, #1).

Research, development, and engineering are needed in the following areas:

- Survey other domains (for example, the computer simulation literature[39]) for approaches or solutions to the semantic composability problem which may be leveraged, adapted, or used as inspiration for solutions in the software understanding domain.
- Study the semantic composability problem with respect to the execution model components discussed in R3, identifying specific research questions and challenges in the software understanding domain.
- Explore verification approaches to the semantic composability problem, enabling emergent properties of software analysis tools to be validated.
- Advance the theory of composing software analysis tools from reusable components. How should those components be organized to promote reusability? What analysis building blocks are most common? What are the needed algorithmic characteristics for addressing the semantic composability problem and which approaches exhibit which characteristics? Are there different categories of analysis where reusability should *not* be attempted, because core assumptions would be incompatible?

---

[39] Claudia Szabo and Yong Meng Teo. "An Approach for Validation of Semantic Composability in Simulation Models." In Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09). IEEE Computer Society, USA, 3–10. https://doi.org/10.1109/PADS.2009.14.

- Study analysis architectures in academic literature and commercial tools to identify approaches which could be leveraged for synthesizing analysis tools from reusable components to a maximal extent.
- Experiment with novel analysis architectures that allow for reusability and composition as a fundamental goal. Develop analysis architectures and components that are highly reusable across different analysis domains and different mission problems.
- Develop approaches to generating analysis architectures from a toolbox of reusable, sharable components. The approach of composing abstract domains, from abstract interpretation, is a promising candidate.
- Develop standards for components to enable rapid assembly of conforming components.
- Research analysis architectural strategies for rapid and effective debugging (see CC4).

### 5.2.3 R2.3 Automated Analysis Tool Synthesis

The full scope of analyses needed across the USG's NS&CI missions is a tremendous challenge. Today, creating new analyses and tools is a time-consuming process, often with years of delay before a new analysis or tool prototype is complete enough to be put into use. This is true even when there is nothing fundamentally novel about the analysis technique—simply applying a known technique to a new problem domain or target software incurs a large cost and delay.

In addition to the compositional architectures of R2.2, another research challenging in accelerating software understanding capability development is to automate the process of creating tools and analyses to the maximum extent feasible, extending the boundaries of what is feasible today. By automating the process of creating custom analyses, the nation can produce software analyses necessary for more mission problems than the nation could with the limited human labor pool the nation has available to do analysis development work. Automation would complement the compositionality approach of R2.2 and accelerate the value gained from compositional analysis building blocks. Although it may not be possible to completely eliminate the human tool developer, the nation should aggressively seek to automate as much of the process as possible to meet analysis demand.

An automated process of synthesizing an analysis tool will instantiate an architecture from R2.2 with selected models from R3 (which were themselves generated using approaches in R4) to gather specific data about the software identify by the mission question decomposition in R7, interfacing with other tools in the ecosystem addressed in R5 to produce the evidence needed by R7.

Research, development, and engineering are needed in the following areas:

- Study the semantic composability problem with respect to analysis tool synthesis algorithms, identifying promising approaches from related fields, defining research gaps, and recommending priorities for exploration.
- Advance the theory of software analysis tool synthesis, innovating algorithms for addressing the semantic composability problem.
- Develop tools that assist in analysis development, making common and repeatable tasks easier. This is a similar approach to what is used in IDE systems like Visual Studio that make some tasks in forward engineering of software much easier but would be applied to analysis creation instead.
- Develop recommender systems (whether rigorous, heuristic, and/or AI/ML based) to suggest collections of components to accomplish specific analytical tasks.

- Create a database of existing analysis components, frameworks, tools, and their analysis characteristics, for use in automated analysis tool synthesis.
- Incrementally advance the art of developing analyses, to automate pieces over time. This includes developing useful helpers in analysis writing that parallel useful code-generation and code-completion systems that are widely used for application development today.
- Engage in human studies of tool developers to inform human-machine interfaces for parsing developer input, presenting information to developers, and enabling developers to rapidly assess and debug faulty tools (see CC4).
- Investigate AI/ML approaches to automated tool generation (building, for example, on research such as Toolformer[40]).
- Research analysis tool synthesis strategies for rapid and effective debugging (see CC4).

#### 5.2.4 R2.4 Designing for Human-in-the-Loop Analysis

The most difficult and time-consuming software analyses today take years to complete. Sometimes, the US Government will spend more than a decade continuously studying a single family of software, to understand it adequately for mission purposes. The bottleneck of the analysis in these cases is not computation, it is the human experts who must work over years to develop expertise in extremely complicated software systems. By specifically targeting tool improvements at helping the human work together with the analysis, research and development effort can make these difficult software analyses more tractable.

Even if R2.1, R2.2, and R2.3 succeed at creating better and more automated analyses, many of the most complicated software systems will defy fully automated, rigorous analysis for many years. The effort of human reverse engineers will be filling that gap in the meantime, but that does not mean nothing can be done on these difficult cases: even when full automation is impossible, tighter integration between human and analysis tool can achieve better results. More research is needed on how human reverse engineers and increasingly automated components can interface in a symbiotic relationship to analyze mission-relevant programs more efficiently.

This research thrust is about automation while *doing the analysis*, as opposed to R2.3, which is concerned with automation in *constructing the analyzer tool*. This research thrust is also related to, but different from, R7. This thrust focuses on analysis architectures to integrate both automation and the human, while R7 focuses on how to present analysis data to the human in a meaningful way.

Research, development, and engineering are needed in the following areas:

- Develop analysis techniques that integrate human reverse engineer feedback into the analysis. Such as: correcting elements the analysis has gotten wrong, updating models of environment/functions by hand, asking the user to solve questions the analysis cannot answer, extrapolating from corrections the user makes to other areas, etc.
- Develop analysis techniques that present useful intermediate analysis answers to the reverse engineer. Such as: which control flow branches can go where, possible values of variables, partial results that are not fully complete yet, etc.

---

[40] Timo Shick, et al, "Toolformer: Language Models Can Teach Themselves to Use Tools," Feb 2023, https://arxiv.org/abs/2302.04761.

- Develop scalable ways for the models of R3 to track data provenance within a single analysis tool, enabling concise statements to be made about what evidence formed the basis of the result.
- Develop approaches for translating, meaningfully summarizing, and presenting complex analysis state of programs for the human analyst, driven by the cognitive processes and knowledge requirements of the human analysis.
- Develop novel analysis designs that integrate human reverse engineer feedback and analysis results in a tight loop, to get the benefits of both. When the analysis is stuck, allow the human to step in and move it forward. When the human is performing rote tasks, allow the analysis to step it and take over automatically.

### 5.2.5 R2.5 Analysis Search Strategies

One way to conceive of software analysis is as a search through a space—there are an overwhelming number of states of a program that could be considered, but not all have to be evaluated relative to a specific analysis question, and of those that do, the order in which they are evaluated can have a dramatic impact on performance[41]. Some analysis architectures avoid any explicit decision about evaluation order, leaving the order to vagaries of the underlying programming language or run-time system; others have an explicit representation of the order of evaluation (such as work queue approach to abstract interpretation used in JSAI[42]).

The exact methods of engaging in a search can have a dramatic impact on the time taken for the search. Choosing a method of search that is ill-suited to a particular problem can elicit worst-case performance, which is often exponential and infeasible for modern programs. Choosing a method of search that takes advantage of heuristics, hints, symmetries, or other elements of the problem/solution space can turn intractable problems into a feasible or even a near-instant one. Thus, the strategies that analysis tools employ in conducting searches in the analysis space are an important element of making analysis feasible.

This research thrust is about the search strategies for *how a tool will decide to explore the search space of a specific analysis problem*. R5.4 involves a similar problem of *conducting a search for which tools and configurations to use to solve a higher-level problem*. These issues are related.

Research, development, and engineering are needed in the following areas:

- Study the state of the art in search optimization as well as how it can be applied to various program analysis problems.
- Identify heuristics that can be used to guide searches through state space search for program analysis problems.
- Develop search strategies that leverage the unique characteristics of program state spaces to make searches more efficient.
- Explore approaches for identifying improved or optimizing search strategies.

---

[41] For example, as seen in the SAT literature: Guo, Mengyu. “A Review of Research on Algorithms for Solving SAT Problems.” 2024. http://dx.doi.org/10.54097/1mn6v127

[42] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

#### 5.2.6 R2.6 Adequate Foundational Tooling for All Target Binaries

Currently, creating an analysis to answer a mission question about target software is a long and laborious process. All too often, it is *almost* an exercise in writing the entire analysis from scratch. In the cases where you have a foundational tooling framework (such as BAP[43], Ghidra's P-code[44], or angr[45]) available for your intended analysis target, you can leverage that framework as a starting point for building the analysis tool. But often that framework was not designed for the situation you want to use it in or requires significant modification to accommodate a new software target. In other cases, the target analysis doesn't have any appropriate foundational tooling, because the mission question being asked or the software you are analyzing are unusual, in which case it may be truly necessary to start from scratch in building the foundations on which the analysis will rest.

Having a richer set of robust, adequate, foundational analysis infrastructure would greatly simplify the process of writing new analysis tools. Instead of building an entire analysis system from the ground up, one could focus on the mission-specific piece and leverage the already-existing analysis foundation for the target software. This would in turn make developing new analyses much faster.

Research, development, and engineering are needed in the following areas:

- Develop adequate foundational platforms for all mission-relevant target languages & software families, one at a time. Here, "adequate foundational platforms" means correctly parsing software artifacts of the chosen language and presenting a basic, flexible, programmatic analysis interface to the analysis developer.
- Study the best existing foundational analysis tools to learn what common features are generalizable and what other unique, target-specific features are necessary.
- Research techniques to build foundational tooling more quickly as new software targets and families come out.

## 5.3 R&D Challenge, R3: Software Execution Modeling

Once suitable, flexible, and composable logics and analysis frameworks are available (see R1 and R2), the next challenge in analyzing software to answering a question about it is to explore the set of possible behaviors for that software, looking for technical evidence of behaviors related to the question. The possible behaviors of software are determined by analyzing the states that it can achieve over the course of its execution.

As briefly explained in FA1, mission-relevant software generally has more possible states than there are atoms in the universe. Dynamic analysis, such as testing or fuzzing, typically explores a miniscule portion of this possible state space; in only very limited cases can the full state space be explored using such techniques. However, the rigor needed for NS&CI missions in the face of concerted adversaries requires analysis options capable of considering this entire state space in some fashion. Transforming the overwhelming search space of software states into a tractable one requires

[43] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: a binary analysis platform. In Proceedings of the 23rd international conference on Computer aided verification (CAV'11). Springer-Verlag, Berlin, Heidelberg, 463–469.
[44] Alexi Bulazel, "Working with Ghidra's P-code to Identify Vulnerable Function Calls," May 11, 2019. https://riverloopsecurity.com/blog/2019/05/pcode/
[45] Shoshitaishvili, Y., Wang, R., Dutcher, A., Kruegel, C., & Vigna, G. (2023). angr: A Powerful and User-friendly Binary Analysis Platform.

aggressively eliding unnecessary detail, while retaining sufficient information to gather the evidence necessary to answer the question at hand.

Currently, the best approaches to this problem appear to be the related concepts of *abstraction* and *modeling*. In this context, abstraction means eliding in a controlled fashion the detail in the data tracked for an analysis, enabling many states of the program to be collapsed together and representing as a single unit for analysis. Modeling means simulating various kinds of program execution (e.g., instruction execution or calls to external functions) in a simpler way than the original. When combined, the models will operate over the abstractions.

All such approaches necessarily involve a reduction in precision, but the concomitant reduction in states is the main benefit. The challenge is in identifying clever abstractions and models which will make the search space tractable while retaining sufficient information to answer a specific mission question on a specific class of programs.

One of the things called for in this challenge is the systematization of the technical details of the various aspects of program execution.

There has been significant academic work in the field of software abstraction and modeling in the past several decades, and several commercial tools built to leverage that work for useful software analysis results. Although progress has been made and these advances are promising, the state-of-the-art tools in industry and academia for software abstraction and modeling have failure modes involving incomplete models or inadequate scaling relative to the full scope of the mission questions and software binaries contemplated in Figure 2. They are, however, a useful starting point from which to build, with many lessons learned that can benefit the efforts included in this roadmap.

### 5.3.1 R3.1 CPU Instruction Modeling

Modeling CPU instructions is the most basic element of modeling software execution. Formal and mathematical precision in how CPU instructions execute is necessary for accurate software analysis but is also surprisingly complicated: modern CPU instruction set architectures can be messy, verbose, and unwieldy. Even when these instruction set architectures are simplified down into a general intermediate representation (such as P-code), the particular peculiarities of the underlying CPU instructions often still shine through as unmodeled instructions or strange instruction patterns that require specific understanding to analyze.

There have been many efforts at modeling CPU instructions for software analysis over the years, many of which are successful and in use today. However, to cover the full national need for software understanding, the NS&CI missions require CPU instruction models that can support combinations of the following: the target CPU modeled, the language that the models are written in (e.g., OCaml for BAP, Python for Angr, Sleigh specs for Ghidra), and the analysis framework into which the models will integrate. Only a very small set of the needed options are currently available.

Research, development, and engineering are needed in the following areas:

- Investigate and develop standardized representations for CPU instruction semantics models to maximize the extent to which they may be reused with an analysis ecosystem (see R5).
- Discover methods to mine CPU architecture manuals to generate or inform the instruction semantics representation automatically.
- Some form of instruction modeling is already done by many existing tools, including compilers, disassemblers, and existing analysis tools; create methods to leverage these

sources for semantic information, translating it into forms reusable by a broader set of tools in the ecosystem.

- Leverage dynamic test environments, including instruction set fuzzers, to inform the generation or validation of instruction semantics models.
- Innovate methods to maximally automate the generation of the transfer function, *f*, based on a given set of state information that must be tracked and a semantics model of the processor.
- Develop methods to vary the precision/abstraction level of instruction semantics modeling (see R1.2).

### 5.3.2 R3.2 Hardware Peripheral Modeling

Hardware peripherals allow software to interact with the physical world through various actuators, sensors, and physical devices, as well as communicate with remote software systems. Although some software has minimal dependency on hardware beyond the universal elements of CPU instructions and memory, there are many important contexts in which tight interaction between the software and mission-specific hardware peripherals is core to the system's intended function; for example, control software of a vehicle that actuates steering or brakes, an industrial control system which senses pressures, temperatures, velocities, etc., and which then manipulates actuators, or a missile flight firmware which interfaces with a radar unit to make decisions on how to manipulate control surfaces to modify flight paths. In these latter cases, modeling hardware peripherals is necessary to answer many mission questions about the system.

Unlike R3.1: CPU instruction modeling, this kind of modeling is about accurately simulating messy and fuzzy physical processes rather than formalized mathematical precision. Simulating those processes often involves statistical models, sampling, and reasonable expectations for sensor distributions.

Research, development, and engineering are needed in the following areas:

- Investigate and develop standardized representations for representing hardware peripheral semantics models to maximize the extent to which they may be reused.
- Study different categories of hardware peripherals relevant to USG missions, including ICS, OT, IOT, and military systems. Sample the expected ranges and distributions of peripheral interactions in normal operation.
- Develop tools with different analysis modalities for "typical" hardware peripheral operation, "unusual" hardware peripheral operation, and "unconstrained" hardware peripheral operation. Where assumptions are made about hardware peripheral operation, make the assumptions explicit and documented.
- Develop automated tools that make modeling new hardware peripherals routine.

### 5.3.3 R3.3 Initial State Modeling

Prior to a program beginning execution, its environment is placed into a particular state, with registers, memory, stack, and other elements initialized in specific ways. When the system has an operating system, the program is also given an extensive initial operating system state, including file handles, external libraries, and process/thread information. In embedded firmware contexts, the various hardware peripherals will also be placed in a particular initial state. This initial state configuration is vital to determining how a program's execution proceeds.

In most current analysis tools, much of this initial state is ignored for simplicity, contributing to both false positives and false negatives. Although not *everything* in the initial state must necessarily be modeled for any given program and question, the *relevant* aspects must be modeled with sufficient precision to gather the necessary evidence to answer the question at hand.

Research, development, and engineering are needed in the following areas:

- Develop machine-readable languages and file formats for defining initial state information suitable for reuse by multiple tools in an analysis ecosystem (see R8).
- Develop reusable methodologies for studying and documenting initial state more rapidly and accurately.
- Study and document the initial state for common program execution environments relevant to USG national security and critical infrastructure missions, including Windows, Linux, and MacOS desktop applications, Windows, Linux and MacOS device drivers and system extensions, mobile apps, embedded firmware environments for common devices, RTOS systems, etc.
- Leverage ML and related data science approaches to mine dynamic execution, header files, and documentation to determine, augment, or validate initial state definitions.
- Investigate approaches for describing initial state elements that are not fixed by a system but are determined by configuration.
- Leverage human factors approaches to create methods for concisely communicating initial state information to human analysts and enabling those analysts to override information in cases where auto-generated models are inaccurate or insufficient.
- Investigate methods for abstraction refinement or lazy instantiation of initial state, keeping initial state information indeterminate until needed by the analysis.

### 5.3.4 R3.4 Memory Use Modeling

A tremendous amount of a program's state space is encoded in the program's memory image. Modeling this state space efficiently and adequately for the analysis is paramount, as enumerating all possible memory configurations naively, even for a single modern program, would take more resources than every computer on Earth put together and more time than the universe has existed thus far.

Although in theory there is an unbounded number of ways of using memory in software, there are numerous design patterns that dominate the vast majority of programs – arrays, structures, linked lists, balanced trees, and more. Entire books are published about well-known data structures with standardized operations and studied properties. Analysis tools can leverage these patterns to simplify the memory models of software. Alternatively, when analysis tools treat all software memory interactions the same and fail to model common data structure patterns, they lose precision and cannot accurately answer mission questions.

Research, development, and engineering are needed in the following areas:

- Research novel approaches to modeling software memory as efficiently and accurately as possible. Separation logic is a promising academic formalism to consider.
- Develop a suite of models for common data structure patterns (e.g. arrays, structs, linked lists, trees, objects.)

- Develop novel approaches for detecting which data structure model to use for a given piece of data. This could include AI/ML based techniques to infer the most likely data structure based on usage patterns.
- Develop novel approaches to compose different memory models together, where a program uses multiple memory usage patterns that are amenable to different models.

#### 5.3.5 R3.5 Indirect, Interrupt, and Exceptional Control Flow Modeling

To analyze a program, one foundational question nearly all software analyses must answer is: after the program executes instruction X, what is the next instruction the program's control will flow to? By answering this question, the analysis can follow the flow of the program and represent cumulative effects of execution over time. This answer to this question, put broadly, is known as a "control flow" of the program.

Often the answer to this question is simple. Most instructions do not have multiple control flow possibilities and will always flow to the next instruction. Some instructions (like conditional branches) can have multiple possible flow targets, but those targets are directly specified and known ahead of time. A few instructions (like indirect branches) do not specify flow targets in advance, instead they rely on the runtime value of program data to determine where to flow next. Determining the possible control flow targets of such an instruction is not trivial and can require an analysis of the program just as complicated as the mission question analysis, just to determine a complete understanding of a program's potential control flow.

Exceptional and interrupt control flows are similarly difficult to reason about. Exceptional control flow occurs when the program performs some operation that raises an exception, like dividing by zero or by accessing inaccessible memory, and normal control flow is suspended while execution shifts to various handlers for exceptional circumstances. Interrupt control flow is similar: sometimes program execution is interrupted by something, perhaps a hardware peripheral, and normal execution is suspended to handle the interrupt.

Accurately modeling all of these different modalities of control flow is simultaneously a very difficult technical challenge and also absolutely foundational to most software analyses. In pathological cases, analyzing control flow is guaranteed to be as difficult as the most difficult questions you can ask about program behaviors. Today, progress is made on recovering control flow despite the difficulty by relying on heuristics, over-approximation, and under-approximation in those difficult cases.

Research, development, and engineering are needed in the following areas:

- Develop models for exceptional control flows: on what instructions are exceptions possible, and where would execution go during and after the exception. The exact mechanics of exceptions vary depending on the Operating System, so multiple models for different OSes will be necessary.
- Develop models for interrupt control flows, based on what kinds of interrupts are possible for a given program environment.
- Research and evaluate methods for resolving indirect control flow. Many different approaches have been prototyped in academic contexts (heuristic methods, over approximations, under approximations, object-oriented virtual method call resolution methods, etc.). Identify limitations of each, opportunities to scale existing techniques, and areas requiring innovation of novel techniques.

- Synthesize the above methods of resolving indirect control flow into a tool that combines the best of each approach and applies it in the situations where it is most relevant.

### 5.3.6 R3.6 Operating System Interaction Modeling

Operating Systems are a common (but not universal) element of the environment software runs in. For most common applications, the OS is the first layer of the program's environment and the only layer that the program directly interacts with. Interactions with hardware, memory allocation, other programs, the network, etc. are usually mediated through the Operating System.

Because the Operating System is such a common and important part of a program's environment, it is deserving of special scrutiny and understanding. Most applications interact with the OS heavily and make strong assumptions about what the OS will do in response. In order to understand and analyze a program, analysis tools must understand and interpret it's intended interactions with the OS environment and how these interactions may lead to unintended behavior.

Research, development, and engineering are needed in the following areas:

- Study individual Operating System environments (such as versions of Windows, Linux, Mac, Mobile OS, RTOS) one at a time, prototyping tools to model those environment interactions and answer specific questions about a program's interactions. The process of developing these tools will reveal both (1) nuances about particular Operating Systems that are vital to model accurately for getting meaningful results and (2) generalizable elements that reappear across multiple software environments.
- Identify intermediate representations for common abstractions applicable across many operating systems, such as files, inter-process communication, network communications, user input/output, passwords, authentication abstractions, web browsing, email, text messages, printing, USB peripherals, wireless communication, etc.
- Develop models for system calls and asynchronous interactions from the operating system to the program, such as signals and exceptions.

### 5.3.7 R3.7 External Library Call Modeling

Very few programs are completely stand-alone. Most modern programs make use of some external code to accomplish parts of their computation. The way in which external code is accessed can vary, but the most common way is through external library calls: calls inside a program that lead into an external library, allowing that library code to perform some function on behalf of the program.

As software becomes more and more complex and feature-rich, modern programs tend to rely on more and more external libraries to get critical computation and interactions done. When analyzing an individual program, the exact code for the external library that will be used during execution is usually not known. In those cases, the analysis must model the external libraries' operations so that analysis of the PUT is accurate.

Most modern operating systems employ some form of shared object so that a program is comprised of its main program and numerous reusable dynamic libraries that are loaded into the program's address space. In some instances, the dynamic libraries may be available to the analysis tool, and in others they will not be.

Research, development, and engineering are needed in the following areas:

- Develop a complete suite of models for commonly used external libraries.

- Develop new categorizations and properties for defining the effects of external library calls.
- Develop new approaches to automatically harvest properties of external libraries by analyzing documentation or online information.
- Develop tools that allow users to inspect and modify library function models as they see fit, to fix any problems that might occur from inaccurate modeling.

#### 5.3.8 R3.8 Execution Concurrency Modeling

Concurrency in software refers to the idea that multiple threads or multiple processes may be running simultaneously in the same space and using the same memory. Mistakes or unintended behavior can occur when there are multiple threads interacting, but the code as written makes assumptions about there being only a single thread of execution. Concurrency has been a long-running source of software bugs, despite continuous efforts to develop formalisms and tools that make software concurrency-safe.

Concurrency is particularly difficult to model because it takes the already enormous state space of a single-threaded program and expands it exponentially by considering multiple possible program states at once. Thus far, there has been no clear, guaranteed, practical way to reduce this expanded state space: every interleaving of program executions from different threads must be considered to understand every possible concurrent interaction. This is an active area of research in academia.

Research, development, and engineering are needed in the following areas:

- Mature existing academic techniques for discovering race conditions in software. Extend them to be applicable to modern concurrent software.
- Develop approaches for reasoning about interrupts in low-level software such a firmware.
- Develop approaches for reasoning about multi-threading, including models for thread creation, locking, semaphores, monitors, and other concurrency programming tools.
- Develop models for common paradigms of concurrency and locking, including verification that the intended paradigm has been employed correctly.

#### 5.3.9 R3.9 Sandbox Environment Modeling

Modern software, especially untrusted software, is often executed inside of a closed environment with limited capabilities known as a *sandbox*. A sandbox is a special software environment which enforces strong restrictions and imposes reduced capabilities on the software executing inside it, thus preventing untrusted software from doing much damage to the system itself. When implemented properly, the limitations of the sandbox prevent the software running inside it from behavior that could be damaging or catastrophic, like invoking system services or interacting with hardware which are outside of the sandbox.

Sandboxes are fundamentally a software security tool and are used to make potentially unsafe software more “safe.” But sandboxes still have several applications relating to the broader software understanding purpose in this roadmap. If a particular application will be run inside a sandbox, that can greatly simplify analysis of it – by adding simplifying assumptions and cutting off analysis paths that will be terminated by the sandbox. Similarly, the software enforcing the sandbox itself warrants special scrutiny – analyzing that software and understanding exactly what limitations it enforces and under what conditions those enforcements are effective, can give mission owners confidence that risks from executing untrusted software have been effectively mitigated.

Research, development, and engineering are needed in the following areas:

- Develop tools and metrics to determine when to use sandboxes – what kinds of programs should be sandboxed.
- Augment existing analysis tools to optionally analyze a sandboxed program – simplifying analysis.
- Research methods to analyze sandboxing software for technical evidence related to validating the sandbox, assessing the limitations the sandbox enforces and the operations may escape the sandbox.

## 5.4 R&D Challenge, R4: Model Generation Techniques

As discussed in R3, accurately modeling various parts of execution is necessary to reason mathematically about large the large state spaces of most software systems. R3 focuses on the different aspects of execution that must be modeled in order to reason about software behavior accurately and efficiently and R2 focuses on how to select and assemble the right models to bear on a system under test. In contrast, this trust, R4, discusses how to rapidly and scalably generate those models with useful precision in the first place.

Whenever a software understanding question needs to be answered for some USG mission, as discussed in FA1, the exact models needed for the software and the models needed for the software's environment are not known in advance. Even if you have a powerful suite of execution models from R3, discovering the correct set of models to apply is a key step in getting an adequate analysis started, and manual mistakes are often made during this step. Without using a set of models that are precise enough to make the mission question answerable, while being abstract enough to make the analysis feasible with limited resources and tailored to the specific system and problem at hand—the resulting analysis will not produce the results the mission requires.

Today, generating the correct set of models for a particular analysis task is a manual process where analysts scour existing analysis methods & software analysis models to find the best fit that already exists, if any; most commonly, however, existing software analysis models are insufficient, so the analyst must adapt existing models or create their own in order to build a sufficient basis for analysis. Even if the research in R3 is wildly successful, there will still be some cases where unique or unusual execution environments (which are relatively common in USG missions) demand bespoke model creation. Making this process as automated and efficient as possible is the subject of this research challenge.

### 5.4.1 R4.1 Rapid Model Generation and Validation

Even with wildly successful research in R3, researchers cannot manually create every single model they will ever need in advance. There are dozens of hardware processors, hundreds of system calls, and hundreds of thousands of distinct library calls in systems relevant to USG missions. Analysis tools need strategies to automatically create and validate models.

To clearly distinguish the challenges here from other thrusts, the analysis tools synthesized by research done in R2 may be comprised of dozens or hundreds of R3 models, which in turn were generated by the techniques described here in R4.

Similar to the issues in FA1, how you construct a given model depends on the question you're trying to answer and the relevance of the model. In some cases, no information relative to the question is handled by the model and the model can be a *nop*. In other cases, the model must be adapted to the specific information being tracked. For example, a timing-based mission question may use a model to report the minimum or maximum execution time; for tracking tainted data, a model may

only need to report the single bit of whether a given output is affected by tainted inputs; for tracking maximum memory pressure, the model may need to report the maximum dynamic memory allocation that may occur.

Research, development, and engineering are needed in the following areas:

- Develop ML-based techniques to rapidly generate models needed in R3, leveraging a wide range of training data (e.g., source code, header files, dynamic test telemetry, social media posts, etc.).
- Research ML-based approaches for leveraging real data to evaluate and improve existing, manually-generated models.
- Develop techniques to generate models of a function from its source code, at varying levels of abstraction/precision (see R1.2).
- Develop techniques that leverage existing documentation to generate coarse models, whether manual pages, header files, etc.
- Identify techniques to generate models that track the particular type of data required by the analysis.
- Develop techniques to leverage fuzzing, instrumentation, and other dynamic techniques to develop a minimum set of behaviors for information model generation and validation. All behaviors observed dynamically should be captured by the model, though in most cases the dynamic behaviors observed will be incomplete.

### 5.4.2 R4.2 Modeling Refinement Strategies

One promising approach to discovering an effective set of models for an analysis is to start with a basic set and refine it over time. An iterative analysis may start with very abstract models, then identify the models that matter most to the control/data flow of the specific analysis question, refining those models to retain more necessary precision. This iterative approach can make initial progress easier and allow the needs of the analysis itself to drive any additional work that may be necessary. The models described in R3 will need to vary depending on the specific information that needs to be tracked, informed by the particular needs for precision vs. abstraction (see R1.2); this research thrust seeks strategies for generating a series of particular R3 models with varying precision needs, and varying data gathering needs informed by the mission question decomposition of R7.

There are multiple existing approaches to refining models that have been used in academic literature, ranging from human-driven experimentation/selection to completely automated Counter-Example Guided Abstraction/Refinement (CEGAR)[46]. Both sides of the spectrum are worth research exploration.

Research, development, and engineering are needed in the following areas:

- Develop techniques to distinguish between what elements of a program must be modeled precisely for the given mission question and what elements of a program can be modeled coarsely, because they do not impact the analysis result.
- Develop tools that allow users to see which models are impacting the analysis heavily and to manually experiment with / refine those models as necessary.

---

[46] Clarke, Edmund, et al. "Counterexample-guided abstraction refinement." Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12. Springer Berlin Heidelberg, 2000.

- Develop techniques to automatically scale the precision and abstraction level of a model, based on automatic analysis feedback or feedback from the user.

### 5.4.3 R4.3 Model Inference for Missing Components

The USG often needs to understand and analyze software where the full environment that software will run in is not available. Sometimes this occurs when the software under analysis is one piece of a networked system where the other pieces are not available (like when analyzing malware); sometimes this occurs when the software can be used in a wide variety of environments or configurations, and the analysts don't know which it will be used in; sometimes this occurs when analysts simply have an executable binary but not all its libraries and dependencies.

Whatever the reason, analysts often need to be able to analyze a system where pieces of the system that the software interacts with are missing. Obtaining analysis results in incomplete scenarios will be essential to many national security and critical infrastructure scenarios. In these scenarios, methods of inferring or filling in the gaps of knowledge are necessary.

Research, development, and engineering are needed in the following areas:

- Innovate techniques to carefully assess available information in the software under evaluation related to the missing components, deriving, inferring or speculating characteristics of the missing components.
- Develop information formats and data exchange standards to document analytic information about missing components, capturing the graduated level of confidence from derived, inference, speculation, or human-assisted approaches.
- Develop methods to infer the content of interactions / network protocol messages that a binary is sending and/or expecting to receive. [cite Caballero paper]
- Research ML approaches to infer characteristics of missing components from the interactions present in the software under analysis.
- Investigate methods to automatically generate speculative models of missing libraries or software components from the inferred characteristics above.
- Develop methods to enable the human analyst to analyst to manually inform or guide inferred models.
- Assemble data sets and develop software understanding-informed features for training ML systems to infer models of missing components.

## 5.5 R&D Challenge, R5: Analysis Tool Ecosystem

R2 discusses research challenges in synthesizing a single analysis tool from a collection of models (discussed in R3 and R4). This section addresses challenges one level above those of the individual analysis tools. This roadmap presumes that, in most cases, running a single tool one time will not produce the evidence needed to answer a high-level mission question (see the discussion for Figure 3). Not only does the high-level mission question decompose into numerous smaller questions (see R7), but for each analysis of those smaller questions constructing the analysis tool correctly to achieve the proper balance of precision and scalability for the particular software artifact under test may require an iterative process (see the "Need for Iteration" discussion in FA1, Mathematical Modeling).

We will illustrate the research challenges with a hypothetical example: a firmware image (P) for a custom controller device must be analyzed to determine whether two particular actuators could

ever be *open* at the same time (Q), which would lead to mission failure. The specific hardware lines leading to the actuators are known from the system design. The CPU instruction set of the firmware is also known.

1) First, it must be determined how the software would command the actuators, and what values constitute *open*. This requires modeling the system hardware that drives the actuators (R3.2), perhaps mining technical spec sheets or other documentation for information (R6), or, as a last resort, calling on the human to identifying the specific technical method for driving the actuators.
2) Once this is determined, an initial analysis tool can be synthesized which will attempt to analyze the target software to determine whether the actuator control methods from step #1 above could ever cause both actuators to be open. Call this tool $T_1$. Note, however, that of the many options of execution models discussed in R3, there is no solid basis yet for deciding which models to use, so a high-level, generic set of models is chosen to synthesize $T_1$.
3) But upon running $T_1$, the answer merely comes back with a very imprecise "*maybe*". To determine where precision was lost in $T_1$, some agent needs to measure it. That agent generates a second tool, $T_2$, which performs the same analysis as $T_1$, but gathers telemetry information about the precision of the values for the actuators and the paths leading to them to determine where precision was lost.
4) The output from $T_2$ illustrates that $T_1$ lost precision almost immediately because the firmware registers various function call-back tasks to be executed based on timers, and the model used to construct $T_1$ abstracted the function pointers of all these tasks to $\top$[47]. $T_1$ was unable to track the specific addresses of any of the main tasks in the firmware, and therefore could not analyze any of them—it had extremely low coverage in its analysis.
5) To proceed, some agent must assess this circumstance and realize that a relatively simple function call-back execution model which tracks all function addresses with full precision, applied to the right addresses in P, will be able to overcome this analysis hurdle. The agent needs to select just such a model from the proverbial toolbox of R3, parameterizing it for the specific addresses in P, and synthesize a new tool, $T_3$.
6) $T_3$ now runs, but still produces the imprecise answer "*maybe*". $T_4$, a telemetry-focused version of $T_3$, reveals that there are numerous other function pointers in P used for control transfers. At this point, a machine-learning based algorithm, run over the telemetry produced by $T_4$, suggests that this binary code may have been produced by C++ using vtables.
7) $T_5$ is then generated using C++ data models from R3, and does a *type recovery* analysis to determine the object-oriented types, type hierarchy, data structures, and more. $T_5$ still produces the imprecise answer "*maybe*".
8) $T_6$, a telemetry-gather version of $T_5$, provides the insight that precision about the actuator state is stored in a particular memory array of one of the object types in the hierarchy, but the memory model used for $T_5$ lacked precision to track the individual values separately.
9) $T_7$ is now generated, using all the knowledge gained earlier, but also imposing a fully precise memory array model on the specific field of the specific object type which is used to track

[47] $\top$ is a mathematical symbol, often pronounced "top"; in this context, it represents any possible value, and therefore a complete lack of precision.

actuator state. $T_7$ is run and produces the output "no", it is not possible for the actuators to both be *open* at the same time[48].

10) The mission owner requires high-confidence and wants a strong evidence package generated containing the detailed technical argument supporting the answer. $T_8$ is generated which emits records of all the data manipulations leading to the actuator states and all the execution patterns controlling those data manipulations.

There are times today when the analysis process is similar to the example above, but with humans replacing many of the $T_x$ tools with manual analysis, the humans replacing the ML tools mentioned in step #6 above, and the humans orchestrating the overall process spanning the steps. There are also times today when a generic tool, $T_G$, written to answer a particular mission question, but not particularly tuned for any give P, is run, and its poor answers are not improved upon due to the cost of doing so.

For clarity, we'll use the term "step" or "tool" to refer to the individual elements listed above, and the term "campaign" to refer to the collection of steps along the journey of answering the mission question, and "orchestration" to the process of determining what the sequence of steps in the campaign should be.

The example above describes one possible potential future. As researchers learn more about the nature of software analysis challenges, this example will need to change. From this current vantage point, the authors can draw some tentative, preliminary observations from this example:

1) **Many types of tools**. There are many different types of tools needed in this software ecosystem. There are tools to reason about software behavior (e.g., $T_1$, $T_3$, $T_5$, $T_7$). There are tools not designed to produce answers to Q but rather to help decide how the next $T_{i+1}$ should be constructed (e.g., $T_2$, $T_4$, $T_6$, $T_8$)—had the CPU architecture not been known, a variety of tools could have provided an initial guess. There are formal tools based on mathematical reasoning, and informal tools, such as the ML tools mentioned in step #6 (and that could have played a far more predominant role than was presented).
2) **Many types of data**. The nation needs tools that analyze very different types of data. Many of the tools mentioned here analyze P. But step #1 could benefit from tools that analyze technical data sheet and other document. Many of the steps involve tools that analyze the telemetry data of other tools. The mission question, Q, considered here is relatively simple to break down and map to specific program behavior; others are not—tools are needed to break down the mission question into the detailed, focused questions needed to guide the analysis of P, as well as to summarize and roll up the evidence data for presentation back to the mission owner (see R7).
3) **Human in the loop**. The human may need to step in at any time, either because a tool is missing, or because the tool is inadequate to the task. For example, today it is nearly always the human who would take the role of manually and incrementally adding telemetry to $T_1$ to determine why it lost precision, the human who would have to recognize the function call-back pattern in step #5, the human who would have to determine that P is a binary built from C++ in step #7, the human who would have to manually construct the evidence package in step #10, and the human who would manually construct all the tools mentioned above. In many cases today, the tools would provide information up to a certain point,

[48] It is important to note that all analysis answers are subject to assumptions and to the accuracy of the modeling. This is not particular to software analysis.

leaving the human to complete the rest of the analysis manually (for example, Ghidra and IDA Pro could be considered tools early in a process which provide the human analyst with some analysis results of the binary and depend on that human to complete the analysis). This research roadmap seeks to reduce costs, improve accuracy, and improve speed by automating as many of these tasks as feasible over time. This will not happen overnight, and so for the foreseeable future the human will be intermingled with the automation. This warrants specific research to minimize the friction of the human interacting with the automation.

4) **An iterative process**. The example above is limited to 10 steps, but the activities represented in steps #3-#5 above could be iterated many, many times: run a tool which can get part of way to the answer, measure where it went wrong, determine what needs to change for the next tool to get further. Perhaps in the future researchers will be able to *a priori* calculate a closed form version of these iterative approaches, but significant research needs to be done on the individual steps of the iterative process first.
5) **The need for a knowledge store**. There are at least three reasons for this. First, some analysis steps in this example may be very expensive to compute. An efficient analysis campaign will cache answers which were costly to acquire, avoiding having to pay the cost again at each step in the campaign. It is not obvious how to do this in many cases, as the context and conditions under which the discovered knowledge hold also may need to be captured, stored, and communicated (e.g., the path condition under which a branch is taken). Second, analysis campaigns should be repeatable. To achieve the high confidence NS&CI missions require, analysis campaigns should be documented in the knowledge store sufficiently for independent researchers to reproduce and verify the results. And finally, a detailed knowledge store for the analysis of other programs in the past could be leveraged by ML systems (see CC2) to make informed predictions about the analyses of the present program under consideration. To facilitate this, the knowledge store could capture the overall steps of the campaign, the recipes and used to synthesize and the configurations used to run each tool, $T_i$, in the campaign, the telemetry gathered for each, the evidence used to determine the next tool in the campaign, $T_{i+1}$, etc. There is too much data to store it all, so research is needed to determine the most useful data to store (see R5.3 below).
6) **The need for debuggability**. This is a complex system with many opportunities to go wrong. Developing such a system will require making debuggability a key goal (see CC4).

The hypothetical example described above, of using a host of low-level tools to answer a high-level mission question is another form of the semantic composability problem described in FA2 and addressed also in R1, R2, and R3. It is necessary to confidently determine that the aggregate, emergent behavior of an analysis campaign will achieve the semantic goals related to answering the original mission question.

The sub-sections below call out several areas of research which are needed to develop a software analysis tool ecosystem and orchestrate an analysis campaign.

### 5.5.1 R5.1 Analysis Tool Characterization

Given a proverbial analysis toolbox, containing many different software analysis tools, selecting the right tool for a given task requires a suitable characterization of the capabilities, requirements, and limitations of each tool. This is true regardless of whether the campaign is orchestrated manually by human analysts, or by an automated system. When the analysis tools in the toolbox do not exist *a*

*priori* but instead of synthesized in response to the needs of the campaign (see R2), the situation is even more complex.

In modern practice today, analysts often have a limited awareness of which tools are best for what analysis job. When understanding every new tool requires significant time and effort, it makes more sense to use tools that are known and familiar, rather than gamble on learning new tools that might end up being less useful. If an immediately useful, digestible, accurate, and trustworthy understanding of the tool space and their characteristics were available, however, then analysts would have a much easier time expanding their horizons to new tools that might suit their current task better.

Research, development, and engineering are needed in the following areas:

- Study the current state of the practice in software analysis by developing taxonomies and surveys of existing software analysis techniques and tools.
- Study orchestration and planning approaches from other domains to identify design patterns, requirements, and other information to inform the development of algorithms for software analysis orchestration.
- Identify characteristics of relevance to software campaign orchestration, including requirements, precision, scalability, and resource usage (execution time, memory, etc.).
- Develop metrics, metrology approaches and algorithms for characterizing software analysis approaches, techniques, tools, or components for cases in which characteristics vary across implementations.

### 5.5.2 R5.2 Tool Configurability, Interfaces, and Interoperability

For analysis tools to be effectively integrated into an orchestrated analysis campaign, separate tools may need to operate in cooperation under the direction of an analysis campaign orchestrator as illustrated in the introduction to R5. The orchestration may need to gather particular, focused information from a given analysis step, requiring the ability to configure that step's analysis tools to achieve a very specific information gathering tasks. Ideally, families of similar tools would have standardized interfaces for their configuration options, facilitating the orchestration process and enabling seamless integration of new tools. Finally, the illustration in the introduction to R5 was described with various tools running serially, but the ability to exploit the parallelism of cloud infrastructure would be highly desirable. This may involve a single tool being multi-threaded, but it may also involve multiple tools dynamically exchanging data in real-time to collectively accomplish more than either could independently.

This research thrust focuses on how the tools themselves need to be designed and architected to integrate into an ecosystem. The next thrust (R5.3) addresses the storage of information produced by the tools.

Research, development, and engineering are needed in the following areas:

- Study the extent to which current tools are developed with hard-coded values and objectives that could instead be made parameters in support of an analysis campaign, producing recommendations for developing tools ready for integration into an analysis ecosystem.
- Research the types of analysis information, context, and telemetry information which could be useful to an analysis campaign orchestration algorithm.
- Investigate data standards and interfaces to capture the information useful to an analysis campaign orchestration algorithm.

- Develop algorithms for analysis campaign orchestration, identifying intermediate information gathering objectives.
- Adapt existing software analysis approaches and techniques to meeting the necessary intermediate information gathering objectives.
- Develop proof-of-concept analysis orchestration testbeds to support extensive experimentation, including focused studies and end-to-end evaluations, leveraging the datasets, metric, and benchmarks of R8.

### 5.5.3 R5.3 Program Knowledge Store

Different tools are effective at answering different kinds of questions about software. Ideally, analysts would use the best tool in the toolbox for different sub-questions, each for the purpose it is most suited to, synthesizing their answers to answer the original high-level question. For example, if the specific task is to determine whether function Y is always eventually called after function X, that might involve a tool that implements a Linear Temporal Logic analysis; if you want to know all the possible values that this JMP RAX instruction might resolve to, that might require a value set analysis over RAX at that address. The answers to these numerous, low-level questions must be gathered together to produce answers to the high-level question about this software (see the discussion of semantic composability in R2 and the hierarchical reasoning in R7).

Unfortunately, this is difficult to do with modern software analysis tools because they are typically unable to export their results to a common format or to accept or integrate data from other tools. Usually, the results of tools have bespoke formats with no easy way to compare or combine them. Enabling two tools to share low-level results for building higher-level results requires significant tool-specific engineering. Consequently, most analysis tools contain their own versions of common, low-level software analysis tasks. This wastes analysis time by duplicating analysis activity. Furthermore, incremental advances in an algorithm in one tool are not readily available to all tools—each would have to reimplement the advance independently.

To enable smaller, reusable components which can leverage incremental advances throughout the ecosystem, researchers must identify ways to share data among components with focused tasks and eliminate duplication. Inspired from the fact databases in Souffle[49], a promising approach is to architect a software analysis ecosystem around a *program knowledge store*. The program knowledge store would be a persistent service for capturing and providing low-level information about the program under evaluation across the lifecycle of a campaign, eliminating the need to recalculate many types of information. For example, if in one stage of a campaign, the orchestration system (R5.4) runs OOAnalyzer[50] to infer the object types and class hierarchies in a binary produced from C++, that information should be cached in order to avoid reperforming that analysis (perhaps multiple times) in later iterations of the campaign. Of course, strategies are needed to handle multiple representations with varying precision of the same data (R1.2). This data store, if designed and implemented with ML requirements in mind, could also provide a novel source of training data for ML techniques (CC2).

Research, development, and engineering are needed in the following areas:

[49] https://souffle-lang.github.io/

[50] Gennari, Jeff. "Using OOAnalyzer to Reverse Engineer Object Oriented Code with Ghidra." SEI Blog, July 15, 2019, https://insights.sei.cmu.edu/blog/using-ooanalyzer-to-reverse-engineer-object-oriented-code-with-ghidra/.

- Study knowledge store strategies from related fields, identifying characteristics and approaches applicable to software understanding.
- Survey existing software analysis tools and the types of data they produce to derive an ontology of the knowledge web that a program knowledge store would need to support.
- Research strategies and configurable policies for storing varying levels of precision of the same data within the program knowledge ontology.
- In many cases, information derived from a program via analysis is only true in a given context, or subject to certain constraints. Investigate and develop standardized representations for describing various contexts, constraints, etc. of the data represented.
- Develop efficient data interchange interfaces and formats for tools in the ecosystem to exchange not only program *facts* and *conditions/contexts* under which those facts hold, but also *data provenance* (where did this data come from), *data confidence* (how certain the analysis is of the data).
- Develop standards for representing analysis knowledge about a program in a common format (inspired for example, by software analysis tools using Souffle and existing standards such as SARIF).
- Investigate architectural designs for a program knowledge store that can serve as an intermediate data repository between different analyses consuming and producing knowledge about a program.
- Research opportunities for ML algorithms to inform program knowledge store designs to enable exploitation of that data to inform other thrusts in this roadmap.
- Explore approaches to leverage the program data store for software analysis introspection, explainability, and debugging (CC4).
- Develop efficient, scalable architectures to support anticipated use cases of a program knowledge store by a campaign orchestration system.
- Innovate strategies for tracking data provenance, uncertainty, and other metadata, identifying useful policies for deriving such information when data is combined.

### 5.5.4 R5.4 Automated Analysis Campaign Orchestration

Currently, analysis campaigns are orchestrated solely by human reverse engineers and analysts. This approach does not scale to mission needs today. Automation of analysis campaigns is needed to achieve the scale of analysis required for the NS&CI mission needs in software understanding. Automation of analysis campaigns may include hybrid solutions in which partial automation assists the human orchestrator (see R2.4) as well as fully automated orchestration.

Research, development, and engineering are needed in the following areas:

- Study options from related fields for addressing the semantic composability problem in software analysis campaigns.
- Evaluate approaches of planning and optimization from related computer science fields (for example, database queries) that may be useful to automated analysis campaign orchestration.
- Study the needs and options for software analysis campaign orchestration by performing initial experiments, retrofitting existing tools with select interfaces and data input/output routines, assessing the utility of the data and prioritizing gaps.
- Develop data conflict resolution approaches to enable an analysis campaign orchestrator to discover and investigate data conflicts and select resolution policies.

- Research approaches to leverage ML and related approaches to orchestrate an analysis campaign.
- Research methods for leveraging human studies and large language models to concisely summarize analysis campaign progress and state at various stages, enabling the human analyst to introspect the data, view summary information, trace decisions back to supporting data, assess orchestration plans, and otherwise superintend an analysis campaign.
- Research methods for leveraging human studies and large language models to actively direct an analysis campaign, by supplying missing information, changing priorities, or overriding decisions.

## 5.6 R&D Challenge, R6: Semantic Knowledge Inference

In the same way that objects in the physical world are all fundamentally composed of atoms, software is fundamentally composed of binary numbers. The composition and relational structure of atoms is what distinguishes radically different physical objects, and in the software world, the composition and relational structure of binary numbers is what distinguishes radically different software artifacts. As discussed in more detail in BR3, to deeply understand and use different physical objects, engineers need more than just an understanding of fundamental atoms: they also need chemistry, biology, material science, and various other higher-level abstractions, to give them the tools they need to ask questions like "will this bridge hold under this weight?". The same is true of software: software analysts need more than just an understanding at the binary level, they also need a series of higher-level semantic abstractions to address mission questions like "does this server contain a backdoor?".

This challenge involves the semantic composability problem discussed in FA2.

Assigning higher-level, semantic meaning to the structures in an executable is technically challenging. The process of constructing binaries involves taking a higher-level representation, often source code, and compiling it down, removing information along the way that is useful for analysis, including type information, data structure boundaries, syntax trees, and much more. There is no guaranteed mathematical way to recover this higher-level information that has been lost—indeed, one individual binary may correspond to an effectively infinite number of different possible original source code programs.

There are imprecise techniques that help in practice, however. Heuristics and inference can provide a reasonable approximation of the source code or other high-level semantic abstractions. The discipline of reverse engineering, practiced by skilled experts, can also produce highly accurate higher-level semantic knowledge about code.

### 5.6.1 R6.1: Computational Heuristics for Semantic Inference

Disassemblers and decompilers bring external knowledge to bear to identify opportunities to apply more semantically meaningful interpretations to these numbers. For example, translating 0x48656c6c6f20576f726c6421 to "Hello World!" involves an inference of an ASCII string and yields a far more semantically useful representation to a human analyst. Currently available disassemblers and decompilers automate some semantic inferences, such as function identification and type identification, but these tools do not provide the full set of information necessary to quickly and easily answer any given mission question.

There are several reverse engineering tools that partially solve the semantic inference problem by deriving a higher-level representation of a binary. For example, Ghidra and Hex-Rays can (with many

limitations) present an approximated source code view of assembly code. Extending these tools would be a valuable way to make semantic inference easier.

Research, development, and engineering are needed in the following areas:

- Defining layers of abstraction above C source code useful for software understanding and decision making (e.g. grouping functions into components of software, analyzing connections and purpose)
- Researching better techniques for inferring types and object hierarchies
- Researching better heuristics for resolving virtual function calls
- Extending RE tool interfaces to support higher-level semantic abstractions

### 5.6.2 R6.2: Human Factors Analysis of Manual Reverse Engineering

The semantic gap left by current heuristic techniques is currently filled by human reverse engineers, manually performing semantic inference, among other tasks. Examples of semantic inference currently carried out by the human include identification of:

- error conditions and paths,
- standard library functions embedded in the system under evaluation,
- common design patterns in data structure layouts (such as linked lists, dictionaries, object hierarchies, etc.),
- common design patterns in code (such as allocators, destructors, iterators, work queues, state machines, etc.)
- the purpose for a given buffer (such as username, password, DNS name, IP address, web URL, etc.),
- enumerated constant recognition based on context (e.g., “6” → “enum ip_proto_tcp” in IP protocol numbering contexts or “enum SPI_FLASH_32KBLOCK_ERASE” in Flash part command contexts)

At present, reverse engineering is more of an ‘art’ than a ‘science’, so it is difficult to say exactly what human reverse engineers do to arrive at their conclusions. However, further research using human factors analysis can elucidate the cognitive processes employed by reverse engineers and support their improvement or automation.

Further research, development, and engineering are needed in the following areas:

- Perform studies to understand and replicate how manual reverse engineers reason about incomplete systems
- Perform studies to understand and replicate how manual reverse engineers climb the semantic ladder
- Develop technical tools/add-ons that can reduce cognitive load on manual reverse engineers and improve their speed
- Develop representations of semantic knowledge that reverse engineers can use to document, track, and share their knowledge (e.g. block diagrams, state machines, decomp, types)
- Design analyses that produce or consume these knowledge representations
- Develop methods to align human generated models with formally verifiable/correct models.
- Identify human interpretable abstractions from a top-down perspective of understanding how humans currently reason about these systems.

#### 5.6.3 R6.3: Semantic Inference Through Machine Learning Techniques

Because semantic inferencing often involves identifying the best conclusion from incomplete data and common patterns seen in other systems, ML techniques are likely to excel. ML trained on labelled software can potentially provide a reasonable estimate of what higher-level semantics are present in an unlabeled sample.

ML has shown the ability to translate between wildly different abstractions levels and structural domains (e.g. summarizing a picture with a few words.) In the field of software analysis, ML techniques may be well-suited to translating low-level information about programs and binaries to a higher-level semantic space in ways that is easily interpretable by the analyst. Existing LLM techniques have already shown impressive results in summarizing code in text for human understanding and even generating code from a textual description. Rigorous application of these techniques to semantic inference problems is likely to yield powerful results.

Research, development, and engineering are needed in the following areas:

- Perform human studies of reverse engineers to identify meaningful high level semantic abstractions as well as the binary features that the human reverse engineers use as clues for making the inference.
- Perform human studies of reverse engineers to identify extra-binary sources of information used by analysts for software understanding and study how experts use those sources to gain insight.
- Develop techniques to mine non-software sources for information about the semantics of structures, interfaces, modules, actions, and design elements of software, and leverage that knowledge for model creation (R3), analysis tool synthesis and configuration (R2), benchmark generation (R8), and analysis campaign orchestration (R5.4), as appropriate.
- Explore the use of ML for automating the inference of high-level semantics from lower-level clues or for mining extra-binary sources of information to gain understanding of the binary under analysis.
- Research ML-based techniques for discovering new Intermediate Representations (IRs) which may be more optimal for human- or machine- based software understanding tasks[51].
- Perform human studies of reverse engineers to evaluate ML-identified features for assisting human reverse engineers at semantic inference.
- Develop software embeddings that preserve high-level semantic features and allow for accurate semantic inferencing

## 5.7 R&D Challenge, R7: Hierarchical Question Decomposition and Evidence Composition

Many NS&CI missions require answers to high-level mission questions about software. Securing port infrastructure, for example, may require an answer to the question "does my port crane software contain a kill switch?" Unfortunately, high-level mission questions like these are technically ambiguous, complex entities with several sub-questions embedded inside them and cannot be directly answered in a single analysis step. For instance, what exactly does the mission owner mean by "kill switch"? What are the many different semantic forms a kill switch might take, and how might

[51] Fawzi, A., Balog, M., Huang, A. *et al.* Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610, 47–53 (2022). https://doi.org/10.1038/s41586-022-05172-4.

analysts prove their existence/absence? Does it count if there is an intended feature for an authorized user to "kill switch" the crane? What if that intended kill switch feature has a vulnerability in it allowing unintended access from third parties? Answers to all these sub-questions (and many more lower-level ones) are needed to synthesize a full answer to the original mission question about port crane software.

Drilling down on the high-level end-to-end process of Figure 3, Figure 6 depicts the notional steps in this end-to-end process, from start to finish.

Research thrusts R1, R2, and R3 focus on providing accurate answers to questions about the low-level behavior of software. These tools are used in Step #4 in Figure 6. However, answers to these low-level questions about the software's operation, possible states, dataflow or control flow, etc., do not directly answer high-level mission questions no matter how rigorously and formally proven. A translation step (or a series of translation steps) from the scope of mission-relevant questions to the scope of focused technical questions about the program is necessary. This is Step #1 in Figure 6.

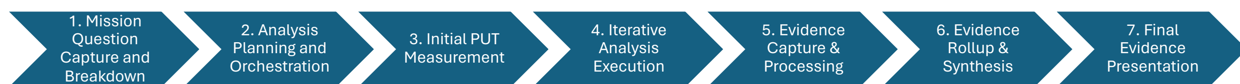


*Figure 6: A notional example of the full, end-to-end process of software understanding. This process starts with a mission question, performs an analysis of the software, and concludes by presenting evidence to answer the initial mission question. Step #4, "Iterative Analysis Execution" is addressed in R5.*

High-level questions must be iteratively decomposed into successively specific, lower-level questions that can be answered by a technical analysis; the authors call this process Hierarchical Question Decomposition (HQD) (see Figure 7 for an example).

Similarly, the technical answers from lower layers in the decomposition must then be synthesized together to answer the next higher-level question, and so forth (Step #6 in Figure 6), until analysts have a final answer to original mission question (Step #7 in Figure 6); the authors call this process Evidence Package Composition (EPC). Thus, the reasoning is hierarchical, first from high-level to low as the question is decomposed, and then from low-level to high as the evidence is "rolled up" back to the top of the hierarchy. Furthermore, the initial and final stages of this overall process today are human-centric, as the Mission Question and final evidence presentation should be in terms most natural to mission owners, while the middle stages must interact with technical software analysis tools. Thus, translations from human-centric to machine-centric forms and vice versa must be performed.

Today, HQD and EPC are performed manually and largely informally by reverse engineer analysts who leverage their personal, often idiosyncratic expertise to understand how low-level technical evidence can relate to high-level mission questions. This process is similar to semantic inferencing (see R6) but goes even further. While semantic inferencing is limited to lifting low-level facts to high-level abstractions *describing the program itself*, HQD and EPC interface with the *questions mission owners have about the program*, an even higher level of abstraction that exceeds any properties of the program itself.

There is no known technical work that has been able to replicate this general manual process automatically, and it is unclear to what extent automation may be achievable. Augmentation of skilled labor may be the best path forward for this research challenge.

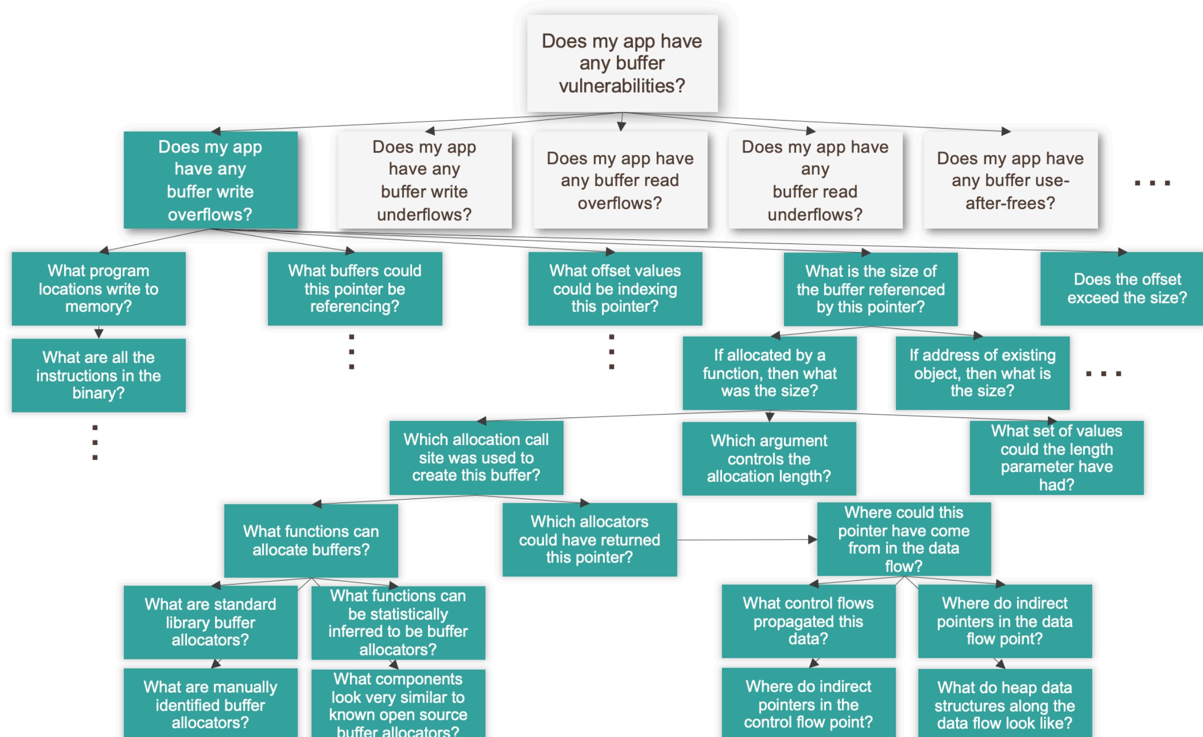


*Figure 7: Notional example of a partial Hierarchical Question Decomposition (HQD). This diagram presents an example of decomposing a high-level Mission Question into smaller, lower-level, more focused questions about the software.*

### 5.7.1 R7.1: Human Factors Studies and Cognitive Mapping

The process and requirements by which HQD and EPC are performed by human reverse engineers and/or analysts is not yet sufficiently understood. The first step towards understanding and improving these processes is careful study, using established Human Factors methods, to identify the cognitive tasks and relationships between them that human reverse engineers employ to break mission questions down into actionable components and synthesize evidence back up to answer the original question.

At the beginning and end of the HQD and EPC processes (steps #1, #2, #6, and #7 in Figure 6), the need for understanding will include both the human analyst and the mission owner. These interactions with the human will be the most high-level and concise, with intermediate steps handling large volumes of low-level information. Translations must be made between human-centric and machine-centric representations to define the mission question and present the evidence gathered.

Research, development, and engineering are needed in the following areas:

- Study human analyst HQD processes using Human Factors methods to understand how human reverse engineers and analysts map high-level mission questions to low-level questions about software behavior.
- Study mission owners and human analysts to identify common goals and decisions associated with mission questions and the information needed to answer the questions, whether through analyst reasoning or automated methods.

- Identify natural and expressive languages for mission stakeholders to use in expressing their Mission Questions about possible software behavior, such that those questions can be readily mapped into machine-readable languages with sufficient precision to guide automated analysis.
- Study EPC using Human Factors methods to understand how human reverse engineers and analysts evaluate different hypotheses, synthesize evidence, and ultimately produce high-level answers out of low-level software properties.
- Produce cognitive maps that describe the landscape of concepts and processes for manual HQD and EPC.
- Identify natural and expressive languages for presenting evidence packages to mission stakeholders which are informative and actionable, identifying useful abstractions and summaries while accommodating detailed questions of providence and certainty.
- Develop recommendations for tooling to augment the human reverse engineers and analysts, reduce their cognitive load, and improve their efficiency.
- Develop approaches to increase automation of the HQD and MQB processes.
- Develop approaches to increase automation of the EPC processes.

### 5.7.2 R7.2: Applications from Formal Software Verification

As stated in the introduction, the focus of this roadmap is on challenges of reverse engineering, not forward engineering. However, there is a thriving field of study centered around formal verification of software requirements and software implementations during software design and development in forward engineering contexts. This work on formal verification of software requirements can provide insight which can be leveraged in reverse engineering contexts. This roadmap can leverage this existing field of work and benefit from inspiration drawn from hierarchical reasoning approaches used in formal verification.

Research, development, and engineering are needed in the following areas:

- Study formal hierarchical reasoning approaches in other formal domains (such as software verification) and assess techniques for applicability to HQD and EPC.
- Develop an ontology of different properties about software, including known categories used in formal methods such as safety properties and liveness properties, and relate these properties to different kinds of software analysis that might be suitable for obtaining answers. There is significant existing work on software properties, but the categories need more granularity.
- Develop conceptual frameworks for categorize or characterizing mission questions, based on what kind of software properties are relevant to answering that high level mission question.
- Develop a clear language and process for translating abstract, high-level questions about software into specific low-level questions about software properties.
- Develop strategies for communicating with mission stakeholders about sensitive or critical information and decisions in software. For example, if a mission owner wants to ask whether an authentication bypass exists in the system, the analysis tools will need to know what “authentication” looks like in this system; knowing whether your critical information can be changed without authorization requires identifying what is “critical” as well as what comprises “authorization”.

### 5.7.3 R7.3: Analysis Evidence and Provenance Collection

Current software analysis techniques are designed to present final answers only, and rarely show any reasoning behind each answer. For example, an analysis designed to find bugs will reveal candidate points that the analysis has concluded are "potentially bugs", perhaps with some additional bug type information, but without any comprehensive chain of reasoning that led to the analysis' conclusion. This stands in stark contrast to how most reverse engineers operate today, where notetaking and evidence collection are paramount for (1) documenting why a conclusion was reached and (2) allowing others to quickly verify and reach the same conclusion.

As discussed in CC4, just as system designers understand how to *design for test* and *design for manufacturability*, so do analysis tool designers need to *design for introspection* or *design for debugging*. That is, researchers must build analysis tools that can be introspected to understand why they go wrong (when they do), why they gave the answer they did, what evidence was used to inform a decision, and how faulty information may influence the output.

In principle, automated analysis systems can also collect this kind of evidence and make it available for users to see. Further research, development, and engineering are needed in the following areas:

- Study the evidence needs of reverse engineers to be able to investigate and keep track of the basis for software understanding conclusions.
- Study the debugging needs of tool developers to be able to identify problems, track provenance of those problems through the tool's computations, correlate intermediate tool information back to the software being analyzed, and identify root causes of failures.
- Develop novel techniques for recording evidence and reasoning during automated analysis, for later presentation to the user.
- Research effective methods to share the provenance of automated analysis conclusions with users. This can be a difficult task because the set of all related evidence for a given conclusion is often too large to represent all at once—it must be filtered or summarized.
- When composing evidence at the conclusion of a campaign (Step #6 in Figure 6 most of the intermediate analysis details used in earlier steps is not needed and can be elided. But mission owners may pose follow-up questions about the provenance or explainability of the answer. In such cases, the analysis campaign system will need to retain adequate information to provide the necessary evidence. For various mission questions, study evidence needs for mission owners to have confidence in analysis results, identifying techniques to guide efficient and effective data retention, at what stages to retain the data, and at what level of precision it should be retained.

### 5.7.4 R7.4: Analysis Confidence Under Uncertainty

There are some software understanding tasks for which complete certainty in the answer is a necessity: for example, "can my nuclear weapon be used without authorization?" must be a certain "no." These cases, for which analysis answers must be provably perfect (or as close to perfect as anyone can possibly get) are called "assurance" cases.

For other software understanding problems, complete assurance is unnecessary—we only need a useful answer, complete with the best available (but not necessarily ironclad) evidence. The authors call these problems "evidence" cases. For example: "this segment of code used in the power grid appears to have a backdoor present" is a powerful analysis result, even if the analysis was only 50%

confident in the answer. It is well worth the cost in such cases to invest in further scrutiny and manual checking.

A critical shortfall arises when the mission warrants an assurance case, but the available capabilities can only provide an evidence case. In such cases, analysis systems need to be capable of reasoning in the presence of uncertainty, and still producing results that have some level of confidence.

Research, development, and engineering are needed in the following areas:

- Study how mission owners make decisions from limited evidentiary bases.
- Review applicable techniques from the field of uncertainty quantification for application to software understanding.
- Developing novel techniques for measuring and tracking uncertainty in software analysis and performing analysis in the presence of uncertainty. This includes measuring and tracking the confidence of analysis results.
- Developing techniques to perform analysis on partial systems, where parts of the software system (such as external libraries or configuration) are unknown.
- Documenting the assumptions made by various analysis techniques and folding those assumptions into analysis uncertainty/confidence.
- Combining the answers from multiple kinds of analysis together on a single question, and deciding which answer is most likely in the presence of conflicting analysis results with varying levels of confidence.

### 5.7.5 R7.5: Artificial Intelligence Approaches to HQD and EPC

HQD and EPC are complex processes which involve fuzzy, not-clearly-defined boundaries, and intelligent decision-making. Thus, these processes are prime candidates for automation via ML approaches.

Existing LLM techniques are effective at reading questions and predicting coherent text responses to those questions. HQD could be framed as a similar text-based problem where high-level questions are phrased as text inputs and the decomposition is a text-based, somewhat formal description of lower-level software properties necessary for the high-level question.

Research, development, and engineering are needed in the following areas:

- Study LLM techniques in understanding human language in HQD and EPC.
- Produce datasets that capture the results of analysts manually performing HQD or EPC, for use in training AI models.
- Extract features from human language that represents high-level mission questions.
- Experiment with training LLMs on software analysis questions and answers.

## 5.8 R&D Challenge, R8: Datasets, Benchmarks and Ground Truth

Research and development programs require some method for measuring progress. In software understanding, progress is measured on the macro scale by showing that capabilities to analyze software for a variety of software understanding mission problems is improving generally over time. Having a method of measuring progress is useful not only for the research program as a whole, but also for measuring progress on individual challenges, comparing different tools, and guiding research towards the most objectively successful approaches. Measuring progress of individual software understanding research problems involves at least three facets: having a question to

answer (whether a high-level mission question, or a very specific low-level question); having a software sample to analyze; and knowing what the correct answer is (i.e. ground truth).

Unfortunately, in software understanding, the phrase "correct answer" can be misleading as there are not always universally-agreed-upon definitions. For example, there are multiple definitions of a *basic block*; consequently, different analysis tools may give different "correct" answers according to these different definitions. As another example, when measuring successful function identification in a binary, should any boilerplate or helper functions inserted by the compiler or linker be included or not? Should interrupt handlers be included? Must all dead code be included? Is a tool "wrong" if it does not include some of these, is it "right" to exclude some, or is there some compromise in the middle? To compare answers objectively, either the "ground truth" representations must reflect a reasonable set of "correct" answers, or the community must standardize their definitions.

Once researchers have a sufficient collection of mission-relevant questions, representative samples, and true answers, they can use that collection to measure the progress of their automated tools. The more closely automated tools' answers are to the correct answers, the more the capability is improving; the more samples automated tools can correctly analyze, the more robust the capability is. In addition, having labeled datasets of software understanding questions and software samples can enable the nation to leverage ML techniques on those questions, if these datasets are built with ML requirements and applications in mind.

The idea of producing a large corpus of datasets to provide a global measure of progress is appealing. However, good datasets and benchmark problems also provide a potential interface between those who  discover a problem and those who may be capable of developing a solution. A team doing research in one area, e.g. semantic labeling, may find that they are frequently hindered by the inability of current tools to resolve common control flow questions. By producing a dataset that is representative of the problems they need solved, whether through a micro-benchmark or larger examples, they open a challenge to the rest of the community. Another team, perhaps working in symbolic execution, may not have realized that certain constructs were a common issue, but by downloading the dataset they have a way to know whether they are making meaningful progress on an important problem.

Currently, large, labeled collections of this kind are hard to come by in the software understanding space. For representative samples, analysts rarely have complete and true answers to software understanding questions documented. Analysts have some answers, particularly for samples that government or other organizations have analyzed before, but even in those cases analysts do not have a guarantee that the answer produced previously is complete and true. For example, one might initially consider a CVE for a piece of software as "ground truth" for the question of "how is this software vulnerable?", but the presence of one vulnerability does not indicate there are no other vulnerabilities present. How could researchers grade tools if they are producing additional results besides the expected one, and no one knows a priori whether those results are correct or not? There is a real sense in which the current solution space is more reflective of the state of tools than the state of the problem. The nation's capability suffers from the streetlight effect— the capability only looks and finds things where it is currently easy to look.

Alternatively, researchers could use non-representative samples (generated code or very simple samples) for which generating true answers is easy, but there is no guarantee that progress on such samples is indicative of progress on the real research challenge. Researchers must be cautious with such an approach because non-representative samples can easily hide the true depth of the

research challenge and trick decisionmakers into going down research avenues that prove ultimately infeasible when applied to real software understanding missions.

Another factor to consider is that automated analysis tool successes may not generalize well to the broad scope depicted in Figure 2 when the tools are highly tailored to specific types of questions, and specific types of software, leveraging non-generalizable heuristics to make progress. This quote illustrates the issue:

> *For better coverage, mainstream tools incorporate heuristics in nearly every phase of disassembly. These heuristics are heavily used in disassembling real-world binaries and, without them, the tools cannot provide practical utility in many tasks.*[52]

Some of these issues motivate the need for R8.2, R8.3, and R8.4 below.

Nevertheless, for some fields of software analysis, limited collections of samples with known answers have been published and used to measure tools[53]. Sometimes these datasets have instigated an explosion of new research and led to tremendous advances in software analysis capabilities[54]. In this thrust, the authors seek to lay the research foundations for datasets that can spark more explosions of research in a variety of software understanding fields.

### 5.8.1 R8.1 Ground Truth Representation Standards

Representing a true answer to a software understanding question can be challenging. Some software questions are ambiguous and admit multiple different answers that are all equally "true." Some software questions require a good deal of drilling down on exactly what the questioner wants in order to come up with a consistent meaning for what the answer should be. Even without these problems, defining a common format to cover a wide range of software questions and software types is not a trivial task.

Even for extremely simple questions, defining a ground truth can be elusive. As an example, given a simple "Hello world!" program, compiled as an x86 ELF binary, eight different tools returned the following number of assembly instructions:

- Radare – 77
- BAP – 105
- Emulated Angr – 28
- Ddisasm – 111
- Objdump – 116
- Ghidra – 112
- IDA Pro – 103
- Binary Ninja - 107

What is the ground truth for disassembly? This is a trivial program with 2 lines of C code, a printf call and a return. Upon manual inspection, all of the tools got it "right". They all had all of the instructions that were part of the main function, but made different choices about how to handle things like linker inserted code, startup code, etc. This is just one example of the kind of ambiguity that can exist in trying to define truth for a given question.

---

[52] Pang, Chengbin et al. "SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask." *2021 IEEE Symposium on Security and Privacy (SP)* (2020): 833-851.
[53] Examples include https://sv-comp.sosy-lab.org, https://taintbench.github.io, and https://nvd.nist.gov.
[54] See https://sv-comp.sosy-lab.org

Research, development, and engineering are needed in the following areas:

- Develop standard, cross-architectural representations for key software understanding program information (e.g., program locations and offsets, data values, control and data flow, symbols, etc.).
- Develop standards of software understanding answers that reflect the underlying ambiguity in ground truth of analysis results, as discussed in the introduction to this section.
- Create standard ground-truth file formats for the datasets in R8.
- Create a library of translation components to read and write these formats, translating from other popular formats.

### 5.8.2 R8.2 High-level Datasets and Benchmarks

One vital element in measuring progress for a software understanding capability is testing that capability on real, high-level mission questions. Examples of high-level mission questions can be found in Appendix A of "The National Need for Software Understanding". The key aspect of these kinds of questions is that they correlate to real mission needs, and if the software understanding capability advances on these questions, it provides strong evidence that the nation is advancing towards exactly the right capabilities. One major downside of using real, high-level mission questions is that they are often ambiguous, making it very difficult to establish ground truth. Given the importance of the many mission questions with national security impact, it is somewhat surprising that a large and broad set of examples is not available to help motivate research and demonstrate the breadth and scope of needed capabilities.

Research, development, and engineering are needed in the following areas:

- Studies of mission question owners and decision makers to understand what types of high-level mission questions would be of interest across various domains.
- Develop a series of high-level datasets, including mission questions, sample programs, and ground truth answers. Each dataset should focus on a single high-level mission question of general interest across government; optionally, each mission question could have a detailed breakdown (see R7), mapping the high-level question about the software into very specific low-level information to gather for each sample.
- Samples should vary across size, complexity, architecture, and expected operating environment; samples should include source code, intermediate compilation artifacts, debug and production versions, and executables for a variety of architectures.
- Curate a high-quality representation of the ground truth for the dataset samples described above.

### 5.8.3 R8.3 Low-level Datasets and Synthetic Benchmarks

There are many different things that need to be modeled to analyze general programs (see R3). A principled scientific exploration involves designing experiment that vary one parameter at a time, controlling for as many other variables as possible. To systematically study software analysis algorithms, researchers need datasets that explore many points along independent axes in isolation. Researchers also need samples that combine these axes in controlled ways. And for all of these, researchers need samples for which the ground truth is known with high confidence. Programs that meet these criteria will likely need to be synthesized.

Static analysis is difficult; analyzing final executables injects uncertainties which compound those difficulties. Consequently, researchers also need datasets that factor out the "binaries are hard" aspects of the challenge. This means the datasets above need to come with a range of artifacts, such as source code, intermediate compiler artifacts, object files, final executables with and without debug symbols, stripped executables, and raw executables such as would occur in a firmware image.

The simplest example of a low-level dataset is a micro-benchmark. Micro-benchmarks attempt to isolate a specific challenge or difficult construct in the simplest possible environment to eliminate other confounding variables.

Research, development, and engineering are needed in the following areas:

- Develop and curate micro-benchmark sets that exemplify specific program analysis challenges with known ground truth.
- Develop techniques to generate larger synthetic program samples which contain specified program analysis difficulties and have known answers to specific software understanding questions. These synthetic programs can form the basis of datasets that test specific low-level software understanding capabilities.
- Develop techniques for measuring select characteristics of existing, mission-relevant software to guide the development of synthetic benchmarks. Such guidance is vital to ensure that datasets are relevant to real-world systems.
- Extend the above techniques to apply to a variety of source languages, like C, C++, Ruby, Python, Rust, Go, and intermediate languages, such as Java byte code, .NET byte code, Javascript, etc.
- Develop a taxonomy for challenges in program analysis, and develop tools to modify individual different axes of software complexity such as: program size, degree of control flow indirection, maximum stack depth, dataflow complexity, degree of heap object usage, etc.

### 5.8.4 R8.4 Dataset Repository

A repository for software understanding is needed to store the outcomes of R8.1, R8.2, and R8.3 and to support other related goals. With some structure, a software understanding repository could demonstrate extreme breadth across both the space of mission questions asked and the space of software under analysis. Such a repository could support the evaluation of tools, serve to motivate research and development, facilitate learning, and build more integrated communities of practitioners.

Datasets need curation, need to be easily discoverable, need structure appropriate to the objectives of the repository, and need to better serve the needs of researchers. One possible standard to consider for managing dataset information is Findable, Accessible, Interoperable, and Reusable (FAIR).[55] Although file sharing and revision control systems (such as Bitbucket,[56] Github,[57], SourceForge,[58] etc.) can be used to store and share files, and afford some ability to tag and organize files, the popularity of custom repositories from other communities demonstrate that simple file

[55] https://www.go-fair.org/fair-principles/
[56] https://bitbucket.org/
[57] https://github.com/
[58] https://sourceforge.net/

sharing tools are insufficient—for example, Kaggle,[59] Hugging Face,[60] the Grand Challenge in biomedical imaging,[61] and Zenodo,[62] a more general repository that seeks to support scientific research in general. Given the scope of the software understanding problem, a repository created specifically to support software understanding research will likely prove quite valuable.

One conclusion from the SUNS 2023 workshop was that datasets and repositories is a top five obstacle in the way of progress, with many researchers expressing frustration at the current state. Different researchers ultimately have different needs, and it may not be possible to meet all needs with a single repository. However, it is likely that many needs can be met.

Research, development, and engineering are needed in the following areas:

- Evaluate popular repositories from other communities to identify principles of success that may transfer to a software understanding repository.
- Engage in human factors studies of various software understanding developer and user communities to identify common use cases which can guide development of a software understanding dataset repository. Such studies would identify needs for ontologies, taxonomies, tags, and other organizational approaches.
- Engage in human factors studies to identify educational gaps that a software understanding dataset repository may help address in developing an expanded workforce to engage in software understanding capability research and development.
- Develop repository options that can not only passively house dataset files, but which can actively run a stock set of available tools on dataset samples to characterize those samples for later search and retrieval.
- Develop repository options that can also actively run stock analysis tools to assess their performance and effectiveness at various software understanding tasks. Such a repository can help measure progress over time (leveraging datasets from both R8.3 and R8.2) and also help identify which tools and approaches are most suitable to different mission questions and problem sets.
- Investigate the use of the repositories listed above as training opportunities for ML-informed recommender systems to help mission owners select the appropriate tool or sets of tools for specific tasks (see CC2). Additionally, investigate the applicability of these tools for informing the analysis campaign orchestration of R5.4.

# 6 Discussion on Prioritization and Sequencing

The roadmap articulated in this document cannot be tackled all at once. Even if financial and personnel resources were immediately available, the authors do not advocate tackling all research thrusts listed above simultaneously. Some research, development, and engineering outlined in this roadmap is more foundational and is critical to explore early, with other thrusts building atop those over time. This section will briefly lay out the authors' considerations on prioritization, sequencing, and other scheduling concerns of the research thrusts.

[59] https://www.kaggle.com/
[60] https://huggingface.co/
[61] https://grand-challenge.org/
[62] https://zenodo.org/

The roadmap can be approached with three different perspectives, each described in more detail below:

1) **Validate the composable modeling hypotheses**,
2) **Start with the foundations**, and
3) **Crawl, Walk, Run**.

These perspectives are not necessarily exclusive.

## 6.1 Validate the Composable Modeling Hypothesis

The financial practicality of solving the software understanding problem requires massive reuse of components designed to be composable. These two points comprise the foundational assumptions upon which this roadmap rests (see FA1 and FA2). Identifying a path forward to accomplish this is paramount to the feasibility of this roadmap. If this can be done, the rest of the roadmap is likely to fall into place, but if it cannot, then the U.S. government may face some incredibly challenging policy decisions about national risk from software.

Therefore, initial investigations should focus on validating FA1 and FA2. The authors recommend validating these assumptions by selecting a small set of proof-of-concept scenarios aligned with Figure 2 to drive initial experiments, not attempting to support all systems or questions of interest but focused on validating the key hypothesis around composable modeling. Executing these experiments will touch on the Cross-Cutting Approaches (see Section 4), semantic knowledge inferencing (R6), and many of the other research, development, and engineering thrusts and sub-thrusts of the roadmap, but far from all. Exploring an end-to-end proof-of-concept of the vision of this roadmap on a small number of more tractable questions and software artifacts will provide valuable early insight which can inform a revision of this roadmap.

The authors recommend that the initial experiment be comprised of three research groups, three mission questions, and three modest software applications from different domains, running on three different systems. Selecting one or two is too few to explore the modular and composable assumptions of the roadmap but selecting four or more is perhaps unnecessary work for an initial evaluation.

## 6.2 Start with the Foundations

In exploring the foundations of software understanding, **R1-R4** should generally come before **R5-R7**. Success in **R1-R4** would provide robust and impactful software analyses tools and building blocks for later challenges to leverage. Success in **R5-R7** would assemble those building blocks in increasingly powerful ways. More specifically, the following research areas are most useful to address early in a research and development program. Three factors were considered in selecting these items for prioritization: their results are foundational and informative to other thrusts, they have known starting points or approaches, and they are more likely to yield nearer-term results. The research, development, and engineering thrusts recommended as priorities are:

- **R8 (Datasets, Benchmarks, and Ground Truth):** The creation of datasets and benchmarks benefits and motivates all the other research areas by enabling systematic and scientific measurements of progress and, consequently, the identification of gaps. In particular, the research thrusts **R8.3 (Low-level Datasets and Synthetic Benchmarks)** and **R8.2 (Dataset Repository)** would help to facilitate the progress of other thrusts by producing datasets to

focus later research and development. These already have viable approaches identified that could be pursued in a short-term time frame.

- **R1.4 (Compositional Reasoning) and R2.2 (Compositional Analysis Architectures):** As discussed in FA2, compositionality is a key facet of developing a software understanding capability that can scale to meet all the needs of the U.S. government depicted in Figure 2. As a result, experimenting with techniques and architectures that promote compositionality is useful earlier rather than later. There are many lessons learned regarding composability in other fields which can accelerate progress.
- **CC4 (Design for Introspection and Debugging) for R2 (Analysis Architectures and Automated Tool Synthesis)**: There are a few open source tools available today which are aligned with early stages of this roadmap. In many cases, scaling those tools up to mission-relevant questions and software involves digging deeply into the tool to understand where it goes wrong. Today, because of the large volume of state involved, this rapidly becomes overwhelming to the tool developer. Developing approaches to facilitate debuggability of software understanding tools during develop is an early investment that will pay dividends during the balance of the roadmap, as well as helping near-term activities improve their tools more quickly.
- **R3 (Software Execution Modeling):** As explained in BR2 (Software Understanding as a Chain-Link Problem), some of the weakest links of software understanding capability relate to modeling software execution. For many tools, when software execution is not modeled adequately, analysis tools lose precision and fail to yield useful results. All of the areas within this challenge are important for some software understanding missions, but R3.4 (Memory Use Modeling) and R3.5 (Indirect, Interrupt, and Exceptional Control Flow Modeling) and R3.6 (Operating System Interaction Modeling) are very common limiting factors in achieving useful analysis results. Shoring up these weak chains in a general, reusable fashion would significantly improve the effectiveness of many software understanding tools.
- **R1.2 (Approaches for Guiding Precision vs. Abstraction Tradeoffs)**: there are many tools that leverage various software analysis techniques to get answers when they can. The scale of the nation's software understanding challenges (see Figure 2) requires scaling to much larger programs while contemplating a broader set of questions than existing tools can support today. Extending existing techniques and innovating new ones are not only essential for long-term success but could also be retrofitting into existing software analysis tools to extend their reach and utility.
- **R2.6 (Adequate Foundational Tooling for Target Binaries):** Currently, the lack of adequate, reusable program analysis frameworks for all target binary formats and CPU architectures of interest presents a substantial barrier to research and the development of software understanding analysis tools. With a moderate degree of research and engineering effort, the research community can begin to address that barrier and accelerate the development of near-term software understanding tools even before the broader aims of the roadmap are addressed. Performing early experiments to inform the development of standard interfaces for such tooling would be an early focus that will make later efforts easier to adopt.

Finally, many of the research challenges in this roadmap start with some form of "Studying" the problem to gain insight. In general, these should be prioritized earlier in the timeline of implementing any roadmap. These "study" activities represent the research needed to understand the particular challenge area well enough to identify key challenges and approaches most likely to succeed.

## 6.3 Crawl, Walk, Run

Once the scope of the NS&CI software understanding problem is understood, the first question typically asked is, "How can the nation do this without having to boil the ocean?" That is, how can progress be made on such a large task as developing a radically improved national software understanding capability, especially given that it is a chain link problem (described in BR2)?

An incremental strategy is needed, starting from the practical limitations of today, leading toward the aspirational future. Metaphorically speaking, this is often called *crawl, walk, run.*

Several of the challenges in this roadmap could be sequenced to lay out a path of incremental improvements, (see the previous sub-section) while many others are interdependent and resist sequential consideration. Even those that can be sequenced are still often quite large on their own. Below, the authors offer a selection of practical strategies for breaking up the larger challenges into smaller, manageable pieces:

- Start by studying the problem itself, not seeking to immediately solve it, but instead to understand it in terms of goals and performing experiments to gather information that may provide insight into how to structure the solution.
- Study related problems in neighboring fields to see how others have solved similar problems.
- Perform controlled experiments, in which as many chain links as possible (see BR2) are "stubbed-out", reducing the overall complexity of the research challenge.
- Varying the complexity of the experiments incrementally, modifying one variable at a time in systematic ways.
- Tackle simpler mission questions first, then move on to more complicated ones. Mission Questions focused on differences between programs may be easier in many ways (e.g., is this program different anywhere other than this one function I think I changed[63]).
- Start with smaller, less complex software samples rather than large, complex ones.
- Start with programs for which source is available, using that source and other intermediate compiler artifacts to accelerate learning about the challenges, the successes, and the characteristics of the approach.
- Instead of attempting to solve a high-level mission question, manually produce a Hierarchical Question Decomposition (R7) and identify a small collection of low-level, focused questions to begin—for example, *What are all the possible values function X might return?* or *What are the possible destinations of this indirect control transfer?*
- Instead of a fully-automated tool, present intermediate results into existing reverse engineering workflows (e.g., push results into Ghidra), enabling the human to interact with the results.
- Rather than attempting to fully automate a challenge immediately, prefer strategies that blend human developers or analysts with the incrementally capable and automated tools.

[63] Draper Laboratory's Comparative Binary Analysis Tool is designed for mission questions like this. https://apps.dtic.mil/sti/trecms/pdf/AD1163850.pdf

## 7 Summary and Conclusions

The challenge of understanding software has been growing for decades and will not be solved quickly and easily. The U.S. government and economy reap trillions of dollars in benefit each year from the use of software but are also increasingly incurring rising costs from behavior in that software which puts U.S. NS&CI missions at risk.

The nation cannot effectively address risks it cannot understand. The roadmap described herein is a first step toward addressing these risks by developing the capabilities necessary to understand the risk to mission arising from its use of software.

The research and development in R1 will strengthen and advance the theoretical, mathematical, and scientific foundations upon which software understanding techniques and tools depend. The architectures and tool synthesis algorithms in R2 will define the structure of analysis tools and the algorithms for generating them. R3 and R4 will produce the reusable software modeling components need by the tools of R2. Individual analysis tools will be developed to work together in an ecosystem of tools which together can execute an analysis campaign through the innovations of R5. The efforts of R6 will enable the models of R3, the tools of R2, and the ecosystem of R5 to work together to infer high-level, useful information from low-level details. This analysis ecosystem will be driven from a hierarchical decomposition of mission questions and produce output to be composed into answers through the advances of R7. And the models, tools, and overall ecosystem will be supported by the benchmarks, datasets, metrics, ground truth, and evaluation strategies of R8.

In addition to the research and development involved in this vision, practical development of mission-impactful capabilities will also require substantial engineering, technical collaboration infrastructure, zero-friction sharing policies, competitive constructs to spur innovation, and top-level government support to embrace technology transitions. Workforces will need to shift as old approaches can be phased out and replaced with more advanced and automated ones. Human capital pipelines will need to be established to develop the talent necessary to discover the innovations needed. International partnerships will be needed to identify win-wins that can bring the best talent in the world together to bear on the shared aspects of these challenges.

Just as the war against cancer could not have made the progress it has through independent, uncoordinated efforts, accomplishing this vision necessarily requires an enduring, coordinated investment, collaboration, and sharing strategy. The interconnectedness of the research thrusts listed in this roadmap is testament to both the barriers which have limited progress to date and the degree of coordination that will be required to fundamentally improve that rate of progress.

Given the predominance of software systems across the globe, the nation that learns best to reason about software through a roadmap like this will dominate global geopolitics for the next century.



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2024-16656R