Verified Trustworthy Software Systems
Problem Book 2024

Brijesh Dongol and Azalea Raad

1 Introduction

This Problem Book aims to present a (non-exhaustive) repository of important verification problems and challenges to the VeTSS community (including academia, industry and the government) and explain why these problems are important. The VeTSS Problem Book is part of our community-building strategy: by identifying important problems, we will develop a community of researchers that cares about these problems and can be encouraged to tackle them. The Problem Book may be used by researchers to make a case to funding bodies. The Problem Book may also help researchers understand, improve and deliver the impact of their existing work, and connect with others working on adjacent topics.

The VeTSS Problem Book differs from state-of-the-art and body-of-knowledge documents on verification. The focus is not on what has been achieved, but on describing what we *want* to achieve, and how we would like to *grow* the field of verification. Other examples of relevant problem books include the NCSC Problem Book (which is a high-level tool to discuss relevant topics within the organisation), as well as the RITICS *Critical National Infrastructure Problem Book* and the RISE *Report on Future Research Trends in Secure Hardware and Embedded Systems*, which are used to direct research in their respective areas. Other documents that readers may wish to reference include *A Path Toward Secure and Measurable Software* report[1] published by the White House and the *Secure by Design: Choosing Secure and Verifiable Technologies* report[2] published by the Australian government.

There is no prescribed timescale for projects (short- or long-term research). That is, impact is important but not immediately needed — a project with longer-term impact is equally good in terms of funding. We push for topics that can be taken up by officials/white papers, provided that the scientific work and the justification are there. However, we cannot advocate for a particular method as it may not work for all set-ups.

A project may pick and choose different areas mentioned in this document. However, we note the following caveats.

1. The Problem Book is certainly *incomplete* and may not cover all verification activities in the UK. It should *not* be used as criteria in the long term, especially because some problems may be missed, or new problems may be introduced by the community.
2. The Problem Book is *not prescribing* a top-down approach and is a *living document* developed *by the VeTSS community for the VeTSS community*. As such, we actively seek input from members of the VeTSS community.

To address the dynamic nature of this Problem Book, we ***encourage community members to submit feedback*** to the e-mail address below (p. 18). Minor modifications will be

[1] https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
[2] https://www.cyber.gov.au/sites/default/files/2024-05/choosing-secure-and-verifiable-technologies.pdf

addressed by the VeTSS Directors, and more substantial changes will be discussed at subsequent Advisory Board meetings.

We present the VeTSS Problem Book in three main parts. First, we discuss the dimensions across which verification problems can be judged, setting out the general scope of the problems within VeTSS (**§2**). We then discuss several important themes within verification that have been identified in consultation with the verification community (**§3**). These themes outline general challenges described for the community, as well as areas within which impact can be delivered. Finally, we highlight potential cross-cutting questions between VeTSS and the other NCSC research institutes: RISCS, RISE and RITICS (**§4**).

2 Verification Dimensions

VeTSS represents a large and diverse community, where verification can often mean different things to different people/organisations. We have identified three key dimensions across which impact can be achieved. In general, the community is united in progressing verification tools and techniques along *each* of these dimensions. This progress enables verification to become more accessible to both specialists and non-specialists by **(a)** simplifying the ease of use and set-up, **(b)** boosting scalability, **(c)** integrating with existing development environments, and **(d)** streamlining the learning curve for (generalist) practitioners.

2.1 Dimension 1: Ease of Use vs Strength of Guarantees

Verification tools and techniques vary widely in how easy they are to use; specifically, whether a tool/technique can be used out-of-the-box as a push-button approach with little to no required training or instrumentation/annotation (e.g. user-defined specifications). Examples of such push-button tools include the influential open-source Infer/Pulse platform developed at Meta and used widely in-house as well as in big-tech companies such as Amazon Web Services (AWS). Similarly, testing techniques are widely used in industrial settings (e.g. to provide statistical/quantitative assurance metrics) as they typically require minimal annotations/instrumentations and do not require specialised training. At the other end of the spectrum lie techniques such as fully mechanised proofs, where a user needs to fully specify the desired behaviours and mechanise their proofs in a theorem prover such as Isabelle/HOL, Lean or Rocq (formerly Coq). Given sufficient resources, such techniques can be used in large-scale settings (e.g. Compcert, CakeML, seL4 and CertiKOS).

The ***scalability*** of testing and push-button tools makes them ideal for industrial settings with large teams of developers who cannot afford the steep learning curve of carrying out mechanised proofs. Typically, however, the ease of use of a verification tool/technique is in inverse correlation with the strength of the guarantees it provides. For instance, Infer/Pulse mostly focus on memory-safety issues (e.g. the absence of null pointer dereferences), and are limited to sequential programs (not accounting for concurrent code). On the other hand, using mechanised techniques one can prove full functional correctness of a given piece of code, albeit at the high time/training cost.

These techniques do *not compete*, but rather *complement* one another, and there is undoubtedly great value in employing diverse tools/techniques spanning this spectrum. For instance, while tools such as Infer/Pulse are highly suitable for large development teams as part of the CI/CD loop in codebases that evolve rapidly, it is more desirable to fully verify (using a mechanised proof) critical software, e.g. a micro-kernel, whose code is not subject to frequent/immediate change. It is also possible to combine these techniques and move from one end of the spectrum to the other to increase assurance.

2.2 Dimension 2: The Compute Stack

A second dimension of verification research is clarifying its area of focus by placing it in the context of the Compute Stack (Fig. 1), ranging from low-level hardware such as logical gates to operating systems and high-level applications. Clear definitions of such a stack enables a separation of concerns at a specific area of interest, supporting modularity. Moreover, it provides a pathway towards *co-specification, co-verification* and *co-design* techniques, where the specification, verification and design of one layer of the stack informs another.

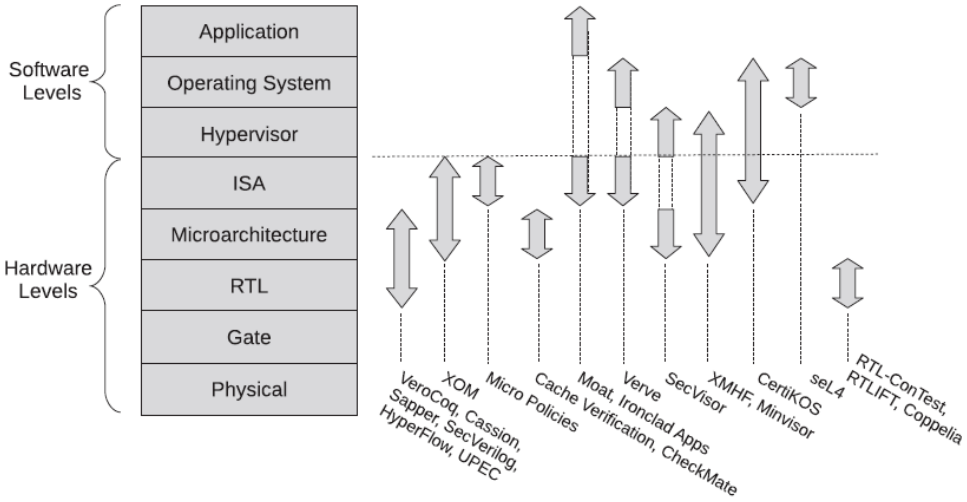

Fig 1. Figure from Erata et al.

Despite many years of progress, verification technology still has a long way to go. Even large-scale projects focus on subcomponents of a system, or a specific (often intricate) aspect, whose correctness may be difficult for humans to judge. Often, one needs to make assumptions about intermediate layers (e.g. the operating system or hypervisor) to enable proofs at higher levels of abstraction. Understanding the gaps allows one to articulate the precise guarantees more clearly and to answer questions such as *the role of a verified component within an unverified system*. Providing precise specifications of the interfaces between different levels of the stack is critical to ensuring that the verified system can ultimately be trusted.

2.3 Dimension 3: Verification Technology Readiness Level (VTRL)

Our third dimension addresses *technology readiness levels* (TRL), which is a widely-used measurement system to assess the maturity of a given technology. Typically, there are nine technology readiness levels, TRL 1 (the least mature, preliminary technology) to TRL 9 (the most mature, market-ready technology). Examples of TRLs include those used at the UKRI (Fig. 2) and NASA[3]. Existing descriptions are, however, inadequate for describing *verification* technologies.

[3] https://www.nasa.gov/directorates/somd/space-communications-navigation-program/technology-readiness-levels/

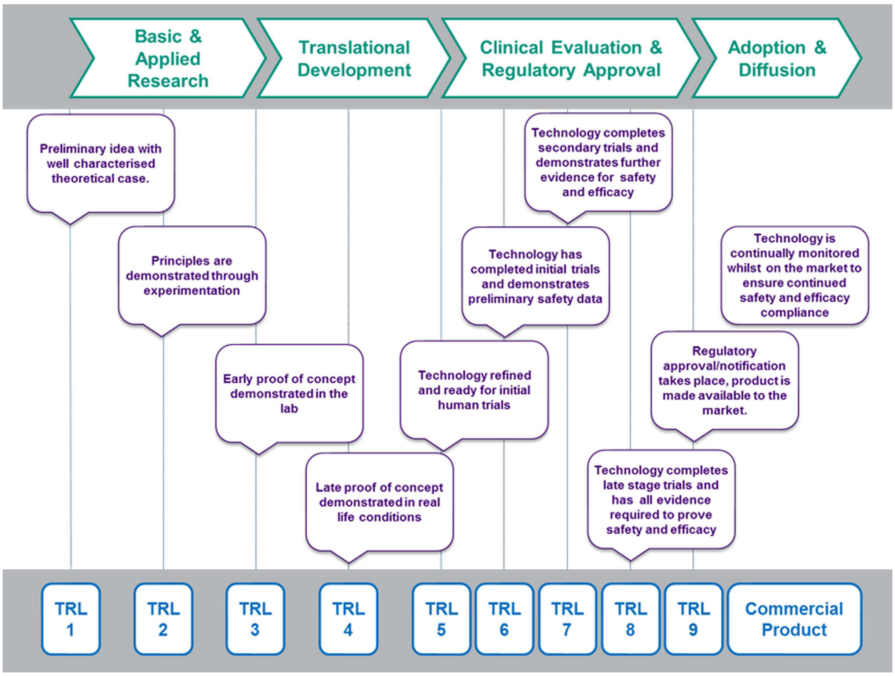

Fig 2. TRLs used by the EPSRC (https://tinyurl.com/UKRI-TRLS)

To this end, we introduce the notion of a *Verification Technology Readiness Level* (VTRL) using a nine-point scale, divided into three phases. Here, one way of demonstrating impact is by progressing a particular technique to higher levels of the VTRL stack, and hence bringing the verification technology closer to wide-scale adoption. However, we emphasise that we see value in conducting research at each of the VTRLs. In general, there are many fundamental open problems that need to be addressed at lower (preliminary) VTRLs, so that they unlock the potential of other techniques at higher VTRLs.

VTRL	VTRL Description	VTRL Phase
1	Proof principles (including logics and semantics)	***Foundational Research***
2	Verification frameworks, proof support and/or compositional reasoning methods	
3	Experimental verification on litmus tests and proof-of-concept examples	
4	Mechanised verification on extended set of litmus tests and proof-of-concept examples	***Translational Development***
5	Mechanised verification on isolated industrial-strength examples (lab conditions)	
6	Automation, reusability and reproducibility on multiple large-scale case studies	
7	Integration with existing verification and/or development environments	***Industrial Deployment***
8	Verification of systems in a deployed (operational) environment	
9	Proof maintenance and robustness of deployed system	

We anticipate different VTRLs to be tackled by different groups of researchers. Levels 1-3 involve foundational work, typically characterising theoretical research and development of proofs-of-concept in academia. Levels 4-6 involve technology-transfer initiatives supported by academia-industry collaborations and large-scale case studies. Levels 7-9 involve development and adoption of verification technologies steered by industry demands.

We note that verification tools such as model checkers and theorem provers are often generic and have the capability of enabling different VTRLs. For example, the Archive of Formal Proofs records a library of proofs for Isabelle/HOL ranging from verified mathematics to industrial-strength applications.

3 VeTSS Verification Themes

We structure the VeTSS Problem Book by defining areas of strategic interest and relevance to industry, academia and the government, categorised as ***themes***. Specifically, we target verification applied to Specification (§3.1); Resilience (§3.2); Protocols (§3.3); Software Systems (§3.4); Programmer/Language Support (§3.5); and Proof Robustness (§3.6). Each theme presents a non-exhaustive list of research topics, challenges and open problems. The purpose of each theme is to cover topics that are of general interest and relevance to the wider community and have been identified to enable work that advances the state of the art across the three dimensions described in §2.

Note that VeTSS themes naturally cover areas such as AI, quantum computing, neuromorphic and biology-based computing without giving these areas specific focus. For example, the use of AI (including synthesis) could apply to (and in fact plays an important role in) each of the themes listed below.

3.1 Specification

Rigorous specifications are at the heart of verification and the key to eliminating ambiguity, providing clarity in the requirements, design and implementation phases of a project. However, this is not always achieved, and overly complex specifications can sometimes become a source of confusion. Formal specifications (e.g. written in temporal logic) may not be understood by non-specialists, engineers or domain experts, making it challenging to validate the correctness of a specification. A specification may also be at the wrong level of abstraction, or be overly verbose, making it difficult to use. Finally, verification frameworks may rely on a *complete* specification, which can be overly time-consuming to develop. A missing ingredient is often appropriately expressive frameworks that provide sufficient levels of mathematical rigour, using domain-specific knowledge as input.

Example Topics

Given a specification, showing that a concrete implementation meets an abstract specification is a fundamental aspect of verification. As such, this theme forms the foundation for most (if not all) VeTSS-related problems, covering both functional and non-functional properties. Some example topics include:

1. *Varying Abstraction Levels* to serve different purposes and audiences, e.g. functional and technical requirements, interface and design specifications.
2. *Domain-Specific Modelling Frameworks* that go beyond existing techniques (e.g. set theory, predicate logic, temporal logic) to describe system properties, behaviours and constraints.
3. *Tool Support* facilitating formal specification, requirements gathering (e.g. requirements engineering) and analysis processes to make formal methods more accessible to developers and engineers.

4. *Translation Tools* that support the formalisation of requirements in different styles (e.g. axiomatic, declarative, operational or denotational semantics) to enable them to be used by different types of program development methods.
5. *Documentation Tools* to generate clear and concise documentation from formal specifications that are usable at different stages of development.
6. *Formal Models of AI* that specify the (potentially quantitative) safety and security guarantees for the AI models in use within a system.
7. *Specification Debugging and Validation Techniques* that allow users to check that their specifications match the real systems that they describe, e.g. by generating test harnesses or litmus tests.
8. *Specification Languages* that formalise guarantees such as concurrency (atomicity), real-time, security (e.g. information flow), autonomy or probabilistic properties of hardware and/or software components.
9. *Specifications of AI* that describe the classification and prediction guarantees of an AI model.

Example Research Questions

- How can we guarantee the correctness/validity of a specification, including when they are developed by different teams?
- Which types of specifications and formal models best support the system development process?
- How can specifications be integrated with existing verification and validation tools?
- How can we describe the relationships between different types of specifications?
- How can we ensure specification integrity, e.g. to ensure that it has not been changed by an adversary.

3.2 Verified Resilience

This theme refers to a system or software component that has been analysed or proven to be capable of withstanding and recovering from various types of disruptions, such as hardware failures, network outages, attacks, or other unexpected events. Such systems may be input-controlled, or more increasingly, autonomous and self-regulating. Resilience covers a system's operations not only during deployment, but throughout its lifetime.

A system may fail under normal use, or due to interference from an active attacker or adversary. Moreover, as more systems use AI as part of their decision engines, verifying the underlying AI may naturally become part of verified resilience. Resilience is particularly challenging in a networked or distributed setting, requiring one to express properties concerning multiple sites and multiple versions (replicas) of data. Here, there must be clear descriptions of robustness and ways to cope with availability and integrity, e.g. in the presence of data disaggregation and sharding. Some domains (e.g. financial systems and healthcare) may additionally require confidentiality and be subject to additional regulatory

controls. Often, resilience is subject to both qualitative and quantitative (including probabilistic) measures.

Example Topics

This theme covers fault and/or failure tolerance of systems (as well as systems of systems) against e.g. natural disasters, cyber attacks and other threats in different domains, including (but not limited to):

- *Critical National Infrastructure (CNI*), e.g. resilience of national power grids, water supply networks and transportation networks.
- *Healthcare Systems*, e.g. patient records and medical devices.
- *Financial Services*, e.g. trading systems, payment networks and data centres.
- *Cloud Services*, ensuring the availability and integrity of hosted data and services.
- *Autonomous Systems*, e.g. autonomous vehicles, robotic systems, satellites and sensor networks.
- *Telecommunication*, ensuring connectivity and service quality against network congestion, hardware failures and other threats.
- *Industrial Control Systems (ICS)*, e.g. manufacturing systems and resilient supply chain management.
- *Outer Space,* including satellites and related systems.
- *Aerospace and Defence*, e.g. mission-critical systems, intelligence and communication systems.
- *Low-power, Embedded and Smart Devices,* including the Internet of Things (IoT).
- *Education/Remote Working*, including online learning and working platforms.

Example Research Questions

- How can we verify resilience, including recovery, degraded services and antifragility?
- How can verification be incorporated into “resilience cases”?
- How can recovery be incorporated into resilience verification?
- How can AI components in resilient systems be verified, or incorporated into, verifying resilience?
- How do future and emerging technologies (including AI) impact this theme?

3.3 Verified Security Protocols

This theme aims to provide assurance that security protocols (including those used for networking and cryptography) and their associated mechanisms are correct. This involves **(a)** ensuring adherence to well-defined security guarantees (e.g. agreement, authentication), and **(b)** providing protection of sensitive information and integrity of data. Proofs may require models of specialised and non-specialised hardware, e.g. Trusted Platform Modules (TPMs), Isolation Engines and Trusted Execution Environments (TEEs).

This theme covers approaches to *describing* and *modelling* protocols (e.g. as Requests for Comments) as well as their security guarantees expressed as (hyper-)properties. Verification may be at the level of a protocol's *design* and *implementation*, which may be performed in tandem so that the analysis of each is informed by the other. This theme also covers theories for verified protocols in systems that have been (or are about to be) deployed, and research aimed at advancing the state-of-the-art theory for modelling and analysing protocols (e.g. to make verification scalable).

Example Topics

The applications of verified protocols are diverse and span multiple domains (where correctness, security and reliability are paramount), including (but not limited to):

- *Secure Communication* between two or more parties over a network, where the messages may or may not be confidential, e.g. over 6G and satellite communication and networking between low-power (IoT) devices at a large scale.
- *Cloud Services*, including their storage, computing power, databases, networking and software components. Examples of cloud service providers include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP) and IBM Cloud.
- *Digital Signatures*, used e.g. for software distribution, financial transactions and legal contracts in order to provide authenticity, integrity and provenance guarantees.
- *Autonomous Systems*, where security plays a key role in resilience against an adversary.
- *Trust Anchors* used by protocols, e.g. cryptographic libraries, secure operating systems and hypervisors and secure compilers and languages.
- *Trusted Computing Modules*, e.g. TPM and TEE, providing hardware security, root-of-trust and remote attestation, as well as secure boot, storage and execution environments.
- *Cryptographic Encryption Schemes*, e.g. TLS/SSH, zero knowledge proofs, key exchange mechanisms, fully homomorphic encryption and key encapsulation.

Example Research Questions

- How can we bridge the chasm between theoretical models and practical implementation of real-world code?
- How can we lower the expertise required for using protocol verification tools, facilitating and increasing their use?
- How can we improve the scalability of protocol verification, e.g. to verify larger, highly stateful protocols?
- How do future and emerging technologies (e.g. quantum computing) impact this theme? For instance, what do we need to do to enable verification of protocols using post-quantum primitives?

3.4 Verified Software Systems

Verification in this theme is primarily aimed at confirming that *software* behaves as specified. Examples include (but are not limited to) functional correctness, memory safety, deadlock and liveness guarantees, and information-flow properties (e.g. non-interference). Depending on the program being verified, a proof may need to specify additional assumptions, e.g. the program's context and/or operating environment.

Verification may additionally be assisted by static and dynamic guarantees provided by a programming language (and compiler), the underlying type system, and/or domain-specific assumptions. Supporting theories may include refinement and abstraction that enable software systems to be developed in multiple stages, where high-level specifications are refined into lower-level implementations. Supporting tools include those that allow natural translation between different semantics, including operational, denotational, declarative and algebraic semantics, as well as the integration of logics and reasoning frameworks, including temporal logics, epistemic logics, Hoare-style logics and separation logics. This theme may also cover the integration of specialised solvers, model checkers and theorem provers for different formal frameworks to support a program development methodology.

Example Topics

Of particular interest are industrially relevant codebases, concurrent programs, correct (de)compilation and co-verification (i.e. verifying hardware *and* software together). Specific application domains include (but are not limited to):

- *Safety-Critical Systems* (e.g. in aerospace and defence, automotive systems, medical devices such as pacemakers and infusion pumps, power grids and transport networks).
- *Financial Services* (e.g. algorithmic trading, risk management and settlement systems) as mentioned above.
- *Co-Verification*, including architectural specifications (e.g. SAIL, ASL), hardware ISAs (e.g. x86, ARM), microkernels and operating systems (e.g. seL4, Linux, certiKOS) and hypervisors.
- *Software Synthesis* involving the generation of verifiably correct executables from formal specifications, including through the use of Large Language Models (LLMs).
- *Verified Compilers* that can provide additional guarantees, e.g. race freedom and memory safety.
- *Verified Decompilers and Lifters* that provide guarantees about correctness of lifting from binaries to an intermediate abstraction (e.g. BIL).
- *Lightweight Formal Methods and Testing Techniques*, including static analysis, data-flow analysis, abstract interpretation, model-based testing, fuzzing that enables rapid checking of type errors, data races and other vulnerabilities.
- *Runtime Verification and Monitors* that can take corrective actions, where necessary.
- *Proof-Carrying Code* to enable programs to formal proofs that demonstrate its adherence to security policies or safety requirements.

Example Research Questions

- How do future and emerging technologies (e.g. LLMs) impact this theme? For instance, how can AI be used to drive verification tools?
- How can we combine verified and unverified components into an integrated system, and what guarantees do they provide?
- What are the underlying software specifications, and how can they be communicated to developers?
- How can verification effectively combine hardware and software guarantees to ensure robust system-level assurances?

3.5 Programmer/Language Support

This theme concerns the development of techniques that enable generalist programmers to integrate (aspects of) verification into daily programming tasks. Programmers may not require full functional correctness of the programs that they write. However, there can be many benefits to supporting lightweight verification or using formal techniques to aid correct-by-construction development, aka *verification for the masses*.

Note that the aim here may not necessarily be to establish full functional correctness, but rather to facilitate eliminating or avoiding common bugs and weaknesses during programming. As such, this theme also covers studies of human-computer interaction, e.g. to ascertain the efficacy and suitability of such techniques, as well as programming languages and compilers that themselves include built-in features for enabling verification and maintainability (e.g. Rust).

Example Topics

The aim here is to provide support during the development process via (lightweight) methods. Examples of such applications include (but are not limited to):

- *Code-Generating Proof Environments*, e.g. Dafny, JML, Why3 and KeY, allowing one to verify programs written in their domain-specific language which can then be compiled to common languages such as Java, Python and C#.
- *Language Integration*, e.g. F*, Liquid Haskell and Reason/ML (Imandra), providing real-time analysis and programming support.
- *IDE Support*, e.g. code suggestions, autocompletion, and real-time error checking, as well as high-quality debugging and profiling tools that support verification.
- *Debuggers and Testing Tools* supporting unit, integration and end-to-end testing; formal bug tracking systems to log, prioritise, and track issues; formal techniques for error handling and logging to capture and report issues; and built-in support for unit testing and property-based testing.
- *Static Code Analysis Tools*, e.g. linters and code analysers to enforce particular coding standards, identify common coding errors and style issues.

- *Auto Generation of Documentation*, e.g. pre/post conditions and inline comments to explain the purpose and usage of code.
- *Language-Level Lightweight Verification,* incorporating contracts to specify expected behaviours that can be validated at run- or compile-time (cf. proof-carrying code).
- *Type (inference) systems* to reduce the burden of explicit type annotations while ensuring type safety, improving code correctness and expressiveness and providing robust error handling mechanisms.
- *Numerical Accuracy* tools such as Herbie that help programmers understand the level of accuracy needed in real-time applications.
- *Concurrency and Parallelism* support, e.g. Erlang's lightweight processes or Go's goroutines, to simplify writing race-free concurrent code.
- *Legacy and Dead-Code Analysis* to identify and eliminate code that is no longer executed or reachable during the program execution and does not contribute to the functionality of the software.

Example Research Questions

- How can we expose the implicit assumptions within a specification or implementation? How can we do this via effective tooling for generalist programmers?
- How can we facilitate adoption of verification technologies in practical development environments, e.g. integrated into CI/CD?
- How can we advance the state of the practice of verification and tooling to maximise the use of limited verification budgets?
- How can we manage frequent and often rapid software changes?
- How do future and emerging technologies (e.g. LLMs and neuro-symbolic proof methods) impact this theme? For instance, how can AI technologies be used for code and proof generation?
- How can one provide guarantees during compilation or code generation? E.g. can we develop compilers that generate warnings when programmers use potentially memory unsafe programming patterns?

3.6 Program and Proof Robustness, Maintainability and Repair

This theme refers to preserving the correctness of programs and proofs in a system under change. This addresses the real-world problem of ensuring that the deployed version of software matches the verified version, where it is important to ensure that the proofs of verified software under change are not rendered obsolete when the software is modified. Real-world software development seldom involves programming from scratch – often the core development effort is on maintaining or modernising existing codebases.

The models and specifications must cover the existing code and their proofs (reference models) as well as the desired programs and their proofs (target models). This enables one to develop techniques for systematically checking that reference models are used when target models are generated. Statistical and ML-based approaches may be used to guide the search process. Other options are developing tools that enable proofs to be maintained by design, as well as integrating these tools in the development process.

Example Topics

The aim here is to support program development. Example include (but are not limited to):

- *Integration with CI/CD Pipelines* including automated (re)verification combined with testing prior to deployment.
- *Interoperability* to ensure compatibility of a verified component with existing ecosystems to facilitate easy integration with other programming languages and libraries.
- *Patch Generation and Synthesis* to identify and generate patches that satisfy specified correctness conditions, as well as non-functional properties, using existing functions, libraries, templates and code fragments.
- *Invariant Synthesis* which may be used to generate missing proof outlines.
- *Code Modernisation, Transformation and Refactoring* to enable improvements to code quality or to address specific issues.
- *Search-Based Program Repair* to explore the space of potential code changes to find a fix, guided by heuristics, mutation operators, patterns, fitness functions, leveraging historical bug fixes and patches from version control repositories and program repair tools.
- *Constraint Solving and Program Repair* to formulate the repair problem as a constraint satisfaction problem.
- *Proof Reuse* that leverages existing correctness proofs or proof fragments to repair incomplete or incorrect proofs, possibly by identifying similar proof patterns or lemmas and applying them in the proof being repaired.
- *Dynamic Analysers* that monitor a program's behaviour during execution and suggest/apply fixes when a bug or defect is identified.
- *Resolution-Based Repair* to resolve formal proofs by inferring correct proof steps based on existing ones, potentially using ML and data-driven approaches.
- *Interactive Proof Repair* that provides suggestions and guidance to the user when automation is challenging, potentially providing the capability of making manual corrections to a proof.

Example Research Questions

- How can formal methods/tools enable rapid proof maintenance or re-verification of software under change?
- How can formal methods/tools enable rapid program repair or patch synthesis for software under change?

- How can the differences across the different versions of programs/proofs be documented in a human-readable manner?
- Can we generate natural language explanations of tool-suggested proof/program repair steps?
- How can we make proofs more readable when required?

4 Cross-Cutting Verification Themes

As well as the themes that fall directly within the VeTSS remit, VeTSS actively aims to address and tackle cross-cutting verification challenges that span neighbouring disciplines, including those prioritised by the other three NCSC (National Cyber Security Centre) Research Institutes (RIs), namely *RISCS* (Research Institute for Socio-Technical Cyber Security), *RISE* (Research Institute for Secure hardware and Embedded systems) and *RITICS* (Research Institute in Trustworthy Inter-connected Cyber-physical Systems).

We discuss several examples of such cross-cutting challenges below.

4.1 RISCS (Sociotechnical Cyber Security)

Understanding the societal, cultural and economical impact of verification is highly important and falls within the purview of RISCS research. For instance:

- How can we integrate humans in the loop for verification? Specifically, how can we expand the “human in the loop” model to embrace ethics, narratives, EDI (equity, diversity and inclusion), human-centred security, trust, privacy and transparency?
- How can we communicate verification to different audiences? This requires an understanding of work habits and culture, their impact and how they may act as barriers to adoption or full implementation of security.
- How much code is verified each day? How can we measure this? How much of the world runs on verified code? Which aspects of code are getting verified?
- How can we drive and foster cultures that encourage such verification measurements (e.g. similar to how the theorem solvers community has benchmarks and competitions to drive research in this area)?
- How can we design surveys to better understand the current verification landscape and the barriers to adoption, e.g. to find out the perceived value of verification where and when it takes place? Why does verification make things better for a business even when nothing breaks? It is alo helpful to understand what verification can do to move the needle, e.g. financial/esteem incentives. For instance, one can ask programmers/companies how much they would pay for formal verification and if they currently pay for it. If the parties involved cannot disclose these figures, then one could ask e.g. for metrics on the number of their staff working on verification.

4.2 RISE (Secure Hardware and Embedded Systems)

VeTSS has a natural link with RISE as the safety and security of software systems ultimately relies on correctly functioning hardware with formally specified guarantees. Initiatives such as DSbD (Digital Security by Design) shows we can do this effectively, and the verification,

hardware, programming languages and security fields interplay naturally. Examples research questions intersecting RISE and VeTSS include (but are not limited to):

- How can we make use of formal ISA models to support verification of critical software such as drivers, firmware, hypervisors and operating systems?
- How can we leverage different types of processing (e.g. CPU, GPU), memory (e.g. NVM), networking (e.g. RDMA) and connection (e.g. CXL) technologies to develop more efficient and more robust systems? This may include future systems such as quantum processors.
- How can we integrate verification with cryptography and post-quantum hardware to provide formally verified security guarantees?
- What are the formal methods to support co-verification, i.e. verification that leverages both hardware and software models?
- How do we model and verify hardware-based AI processors (e.g. neural processing units)?
- Can we achieve full-stack verification of embedded systems with limited functionality?

4.3 RITICS (Trustworthy Inter-Connected Cyber-Physical Systems)

Verification has long been recognised as being of importance to guarantee the correctness of critical services such as manufacturing and transport, energy, water and telecommunication networks. Such systems are often cyber-physical in nature and contain (autonomous) networked computers that can control physical systems. Examples of research questions intersecting RITCS and VeTSS include (but are not limited to):

- How do we model and verify cyber-physical systems that (autonomously) monitor inputs from sensors, control outputs to actuators, implement logic and arithmetic operations and manage communication with other devices or systems? Systems may be connected to form a large network or a swarm.
- What are appropriate formal methods for scalable verification of industry-standard frameworks for PLCs such as ladder logic?
- How can we model and verify bespoke communication protocols between cyber-physical systems, particularly when they must provide security guarantees and protection from hardware and software attacks.
- What is the role of formal methods and verification in the development of digital twins, where we require *computational* models and specifications that simulate the behaviour, characteristics and interactions of their counterpart physical systems?

5 Acknowledgments

This document was prepared in collaboration with input from the members of the *VeTSS Advisory Board*, a panel of *Expert Reviewers* and the *VeTSS program managers*.

VeTSS Advisory Board: NCSC, Jade Alglave, Rob Ashmore, Sofia Guerra, Ekaterina Komendantskaya, Brad Martin, Alastair Reid, Peter Sewell, Gregory Smith, Greta Yorsh

Expert Reviewers: NCSC, Rob Ashmore, Ana Cavalcanti, Chris Hankin, Matthew Hill, Steve Schneider, John Wickerson

VeTSS Program Managers: Teresa Carbajo-Garcia and Ling Zhang

Contact Address:
contact@vetss.org.uk (Feedback welcome)