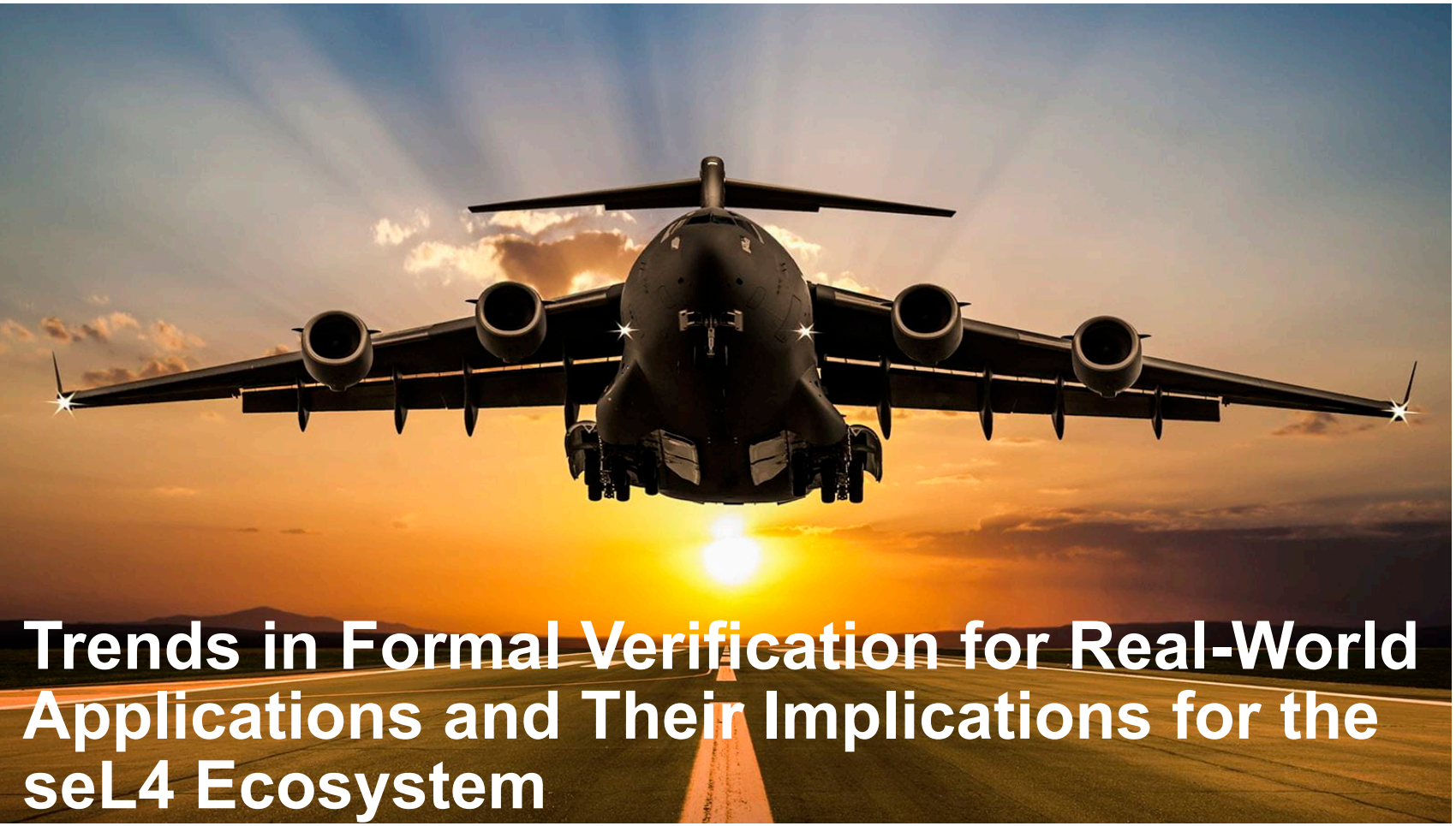


Trends in Formal Verification for Real-World Applications and Their Implications for the seL4 Ecosystem

David S. Hardin, Ph.D.
Applied Research and Technology
Collins Aerospace
david.hardin@collins.com

Disclaimer

The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense, the U.S. Government, Collins Aerospace, or RTX.

No Generative AI systems were harmed or exploited in the creation of this presentation.

Background

- I have been involved in several product development and accreditation/certification efforts over the past 20+ years at Collins Aerospace that feature Formal Verification (FV):
  - A microprocessor with a verified separation kernel in microcode (AAMP7)
  - Type-1 Crypto devices (e.g., Janus)
  - High-Assurance Cross-Domain Systems (e.g., Turnstile)
  - Others I can't (yet) name

- I edited a book in 2010 based on some of those earlier experiences, as well as other emerging FV research at the time that has since matured nicely, including seL4

- Over the years, the capability and accessibility of formal tools has grown — now, one can now routinely prove properties about real-world engineering artifacts, something that required heroic effort 15 years ago

- I am currently engaged in a DARPA program, PROVERS, whose aim is to bring formal verification to non-specialist developers, especially in systems engineering

Introduction

- Formal Verification is “hot” currently as it has not been in several years
- To many, FV is seen as the “silver bullet” that will save Generative AI from its hallucinatory tendencies
- A number of well-funded startups have emerged that seek to utilize Generative AI techniques to produce formal proofs expressed in Interactive Theorem Proving (ITP) frameworks, especially LEAN
- But there are subtler forces at work bringing formal verification to the fore, and not all of these have a lot to do with Generative AI
- We will explore some (not all) of the trends in FV in this talk, and relate them to the seL4 ecosystem and community

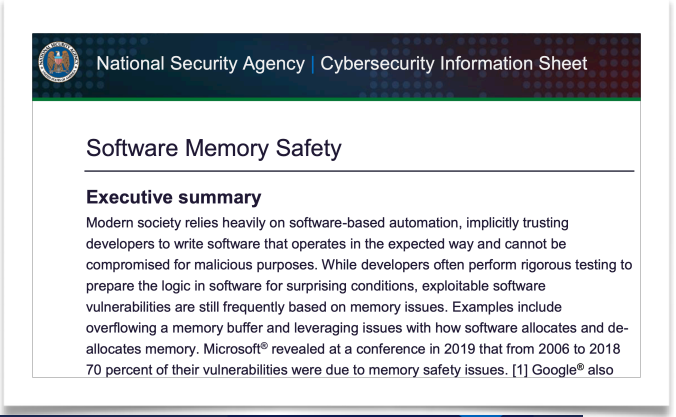
Trend 1:

Memory-Safe Language Adoption

Trend 1: Memory-Safe Languages Gain Mindshare

- We were facing an increasingly bleak future, in which basically un-analyzable, unverifiable C++ code was propagating unchecked. But the tide has started to turn:

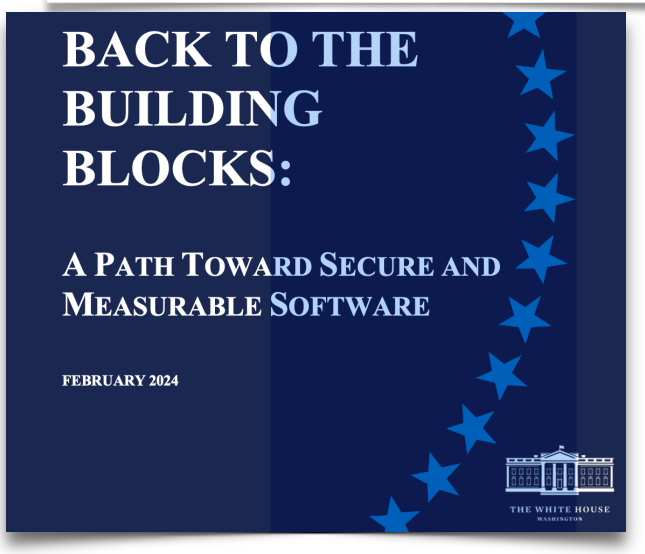
- "NSA recommends using a memory safe language when possible." (Nov. 2022)
- The White House published a report championing the adoption of memory safe programming languages to enhance software security (Feb. 2024)
- Microsoft, Google, and Amazon have all announced significant Rust initiatives
- Rust is supported in the Linux kernel
- ***Memory-safe language requirements are beginning to appear in U.S. Government contracting***


National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also



Why Memory-Safe Languages? Why now?

- Memory-safe languages are not new:
  - For example, Collins has successfully used Ada in major commercial and government avionics dating back to the 1980s, and continuing today
  - Collins used SPARK effectively on high-assurance products for the intelligence community in the 2000s

- Recent improvements in compiler technology have made memory safety very low cost
- Additionally, novel memory ownership models (e.g, in Rust) have allowed references to be used safely
- Development organizations have tired of continual memory errors, resulting in a never-ending parade of security vulnerabilities, despite the use of increasingly sophisticated analysis tools

The Rust Programming Language

- Rust has several assurance advantages over C/C++, including:
  - Improved type safety
  - Vastly improved memory safety
  - No arbitrary pointer arithmetic
  - **…in short, 80% of C/C++ security flaws are eliminated outright!**
- Rust supports modern programming idioms such as a match primitive, traits, immutability by default, etc.
- Basic Rust syntax is familiar to C/C++ developers, easing the transition
- The Rust compiler produces code which is competitive in speed to C/C++
- As of July, 2026, Rust has climbed to the Top 10 in the TIOBE Programming Community Index

A Remarkable Statement from Kent Overstreet, Linux bcachefs lead:

“Rust is going to mean a codebase that's much less fragile, more stable, and easier to work on, and it'll also make it much easier to bring in younger engineers - we don't have as many people learning C anymore! Even better, it opens up full formal verification: with Rust handling memory safety, the rest becomes tractable, in a real production codebase. And we want that: filesystems are full of invariants that today are checked by debug assertions, but actually exercising those debug assertions in testing requires massive amounts of hardware and testing, and many hours of waiting on the CI. A fully formally verified filesystem isn't going to happen overnight, or for many years, but being able to start to apply formal verification to the parts of the code that really need is going to result in real improvements to reliability, and soon, and make life so much easier. It'll change how we do engineering.”

https://evilpiepirate.org/git/bcachefs-tools.git/tree/Changelog.mdwn

17 June 2026

Verus — Rust Code Verification

- Verus is an open source Rust code verification environment under development by an international team led by Carnegie Mellon University
  - Industry users include AWS, Microsoft
- Verus has been utilized in a number of operating system, concurrent data structure, and distributed algorithm verification efforts
- Verus utilizes Rust syntax to express precondition and postcondition annotations, loop invariants, etc.
- Verus employs an SMT solver backend to prove “ensures” postconditions, given “requires” preconditions

```
fn timeTriggered<API: Manage_Heat_Source_i_Full_Api>(&mut self,
                                                     api: &mut Manage_Heat_Source_i_Application_Api<API>)
  requires
     old(api).lower_desired_temp.degrees <= old(api).upper_desired_temp.degrees
  ensures
    // guarantee lastCmd
    // Set lastCmd to value of output Cmd port
    (self.lastCmd == api.heat_control)
  && // case REQ_MHS_1
     // If the Regulator Mode is INIT, the Heat Control shall be set to Off.
     ((api.regulator_mode == data::Regulator_Mode::Init_Regulator_Mode) ==>
      (api.heat_control == data::OnOff::Off))
  && // case REQ_MHS_2
     // If the Regulator Mode is NORMAL and the Current Temperature is less than
     // the Lower Desired Temperature, the Heat Control shall be set to On.
     ((api.regulator_mode == data::Regulator_Mode::Normal_Regulator_Mode &&
       api.current_tempWstatus.degrees < api.lower_desired_temp.degrees) ==>
      (api.heat_control == data::OnOff::Onn))
  && [...]
{
  // -------------- Get values of input ports ------------------
  let lower: data::Temp_i = api.get_lower_desired_temp(); // gives lower <= api.upper_desired_temp.degrees
  let upper: data::Temp_i = api.get_upper_desired_temp(); // gives api.lower_desired_temp.degrees <= upper

  let regulator_mode: data::Regulator_Mode = api.get_regulator_mode();
  let currentTemp: data::TempWstatus_i = api.get_current_tempWstatus();

  //================ compute / control logic ============================

  // current command defaults to value of last command (REQ-MHS-4)
  let mut currentCmd: data::OnOff = self.lastCmd;

  match regulator_mode {
    // ----- INIT Mode --------
    data::Regulator_Mode::Init_Regulator_Mode => {
      // REQ-MHS-1
      currentCmd = data::OnOff::Off;
    },
    // ------ NORMAL Mode -------
    data::Regulator_Mode::Normal_Regulator_Mode => {
      if (currentTemp.degrees < lower.degrees) {
        assert(api.current_tempWstatus.degrees < api.lower_desired_temp.degrees);
        // REQ-MHS-2
        currentCmd = data::OnOff::Onn;
      } [...]
    } [...]
  }
}
```

Trend 2:

Mathematicians Adopting Theorem Provers (Finally)

Mathematicians and Mechanized Theorem Proving

- Mathematicians have been historically reluctant to use theorem proving tools, due to surveyability concerns, unfamiliar theorem prover languages, the tendency to get dragged “into the weeds” of endless low-level lemmas, as well as a lack of foundational mathematics libraries
- Starting in 2017, Kevin Buzzard began a project at Imperial College London to develop a comprehensive mathematics library for the new Lean theorem prover, in a classic example of “Build it and they will come”
- The mathlib effort has subsequently grown to some 760 contributors, and has attracted the attention of such notable mathematicians as Fields Medal winner Terence Tao
- The success of Lean + mathlib in the mathematics community is somewhat due to the concurrent rise of Generative AI, which has ameliorated problems with unfamiliar notation, as well as the “endless detailed lemmas” issue
- *Why this matters outside Mathematics*:
  - Based on the success of mathlib, there is significant momentum to develop CSLib, a similar set of libraries and tools for Computer Science and related fields
  - Endorsement of famous mathematicians like Terence Tao is a significant feather in the cap of formal verification

Trend 3:

Advances in Mechanized Theorem Proving

Sample Advances in Mechanized Theorem Proving

- Proof Engineering is increasingly recognized as a valued discipline, thanks to the efforts of Gerwin Klein (Proofcraft), Eric Smith (Kestrel), and other researchers who wrangle very large proofbases on a daily basis

- Major releases from many interactive theorem provers, including:
  - ACL2 8.7 was released in March 2026. ACL2 includes a highly accurate x86 model that can boot and run Linux at a simulation speed of 3.3 million instructions per second, all hosted on Common Lisp
  - HOL4 Trindemossen-2 was released in August 2025
  - Isabelle2025-2 was released in January 2026
  - Lean 4 has accelerated its release schedule, with minor releases occurring monthly.

- SMT solvers are adding capabilities from interactive theorem proving, such as induction

- The SMT solver cvc5 is adding first-order logic proof generation capability, allowing SMT proofs to be checked by other, trustworthy tools

- Researchers are making progress on the SMT “brittleness” problem, in which seemingly minor changes to the source code undergoing verification can cause a previously proved result to fail

Advances in Mechanized Theorem Proving (cont'd.)

- Recent verification-enhanced programming language tools such as Rust/Verus have made great strides in software verification by non-specialist developers
  - Verus annotations use Rust syntax, easing developer ability to specify formal properties
  - The Verus engine takes advantage of the Rust memory model, and technical issues such as separation logic are not exposed to the user

- SMT-based tools (e.g., Verus) are developing bridges to interactive theorem provers (e.g., Lean) on the backend when SMT proofs do not succeed
- Interactive theorem provers are able to make use of SMT solvers on the back end

- The Galois crux-mir symbolic verification tool allows one to run a Rust program to a starting point, capture the MIR intermediate state, then feed this initial state to a symbolic execution engine to perform verification to a defined end point

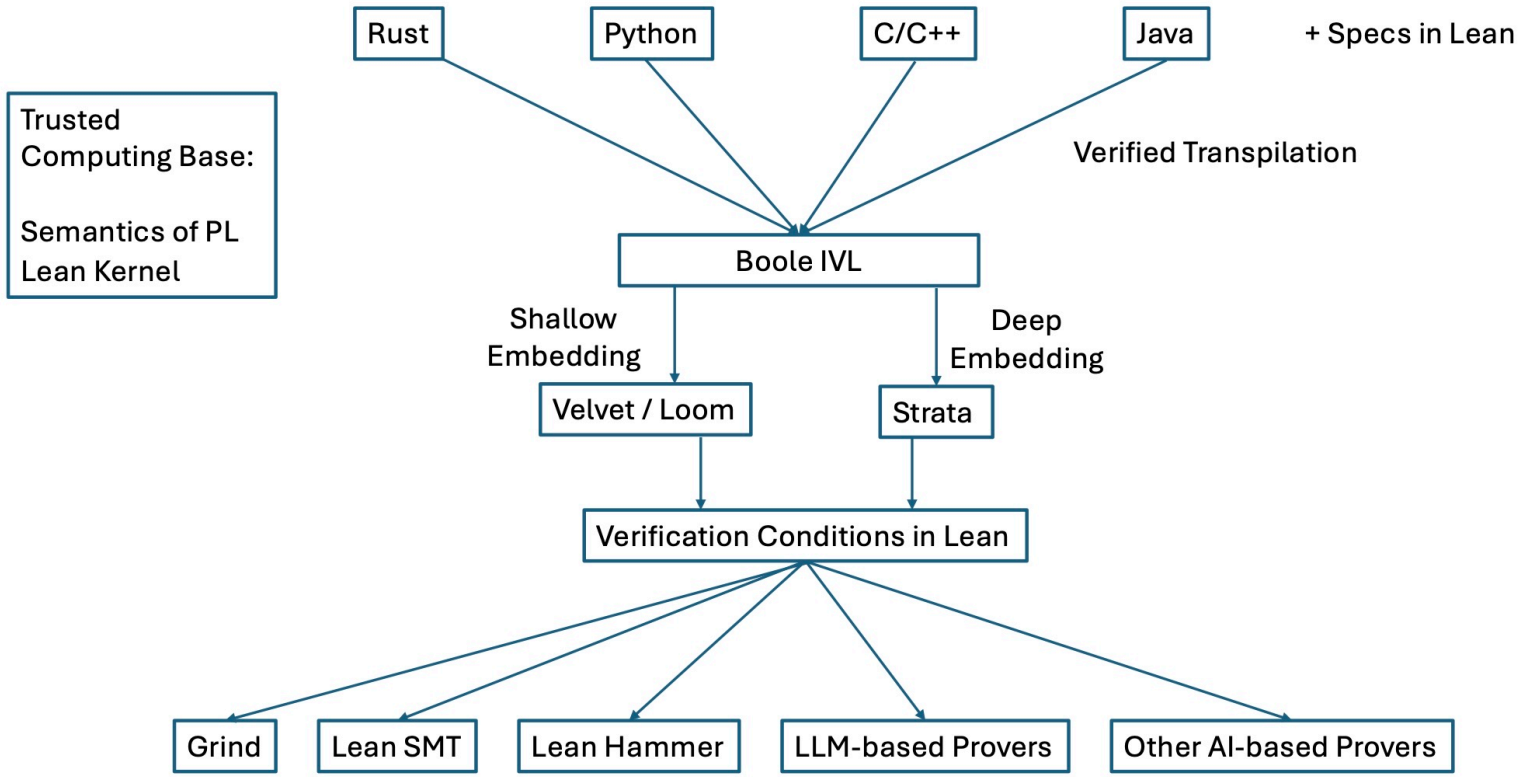
Advances in Mechanized Theorem Proving (cont'd.)

- Generative AI/Theorem Proving Mashups
  - LLM's have been successfully utilized to provide tactics for tactic-based theorem provers to complete "proof sketches"
  - LLM's have been used to translate textbook proofs into proofs in particular interactive theorem proving systems
  - The Lean team has utilized an LLM to find bugs in Lean itself, resulting in several proofs of false. These have been fixed in the most recent version of Lean

- CSLib Development
  - CSLib is "an open-source framework for proving computer-science-related theorems and writing formally verified code in the Lean proof assistant. CSLib aims to be for computer science what Lean's Mathlib is for mathematics."
  - CSLib's steering committee is led by Clark Barrett of Stanford University, and recently attracted funding from AI philanthropy organizations

CSLib Envisioned Structure

https://arxiv.org/pdf/2602.04846



Trend 4:

Generative AI for Formal Verification

A Sampling of Generative-AI-for-Formal Verification Use Cases

- LLM's have been successfully utilized to provide tactics for tactic-based theorem provers to complete "proof sketches"
  - This was the Collins FV team's first successful LLM use case, back in 2023
- LLM's have been used to translate textbook proofs into proofs in particular interactive theorem proving systems
  - Sometimes by literally scanning parts of the physical texts
- The Lean team has utilized an LLM to find bugs in Lean itself, resulting in several proofs of false. These have been fixed in the most recent version of Lean

DARPA PROVERS

Pipelined Reasoning Of Verifiers Enabling Robust Systems

• Develop automated, scalable **formal methods** tools that are integrated into traditional development pipelines using "proof engineering" techniques

• Enable traditional product engineers to produce and maintain **high-assurance** national security systems

• Collins-led INSPECTA team is **making formal verification** across the entire software development stack **accessible to non-formal methods experts** through automated analysis, DevOps integration, and improved user feedback

- Key Results: SysML v2 System Modeling Language support with assume/guarantee contracts, as well as code generation for modern memory-safe languages, specifically Rust, with Verus annotations

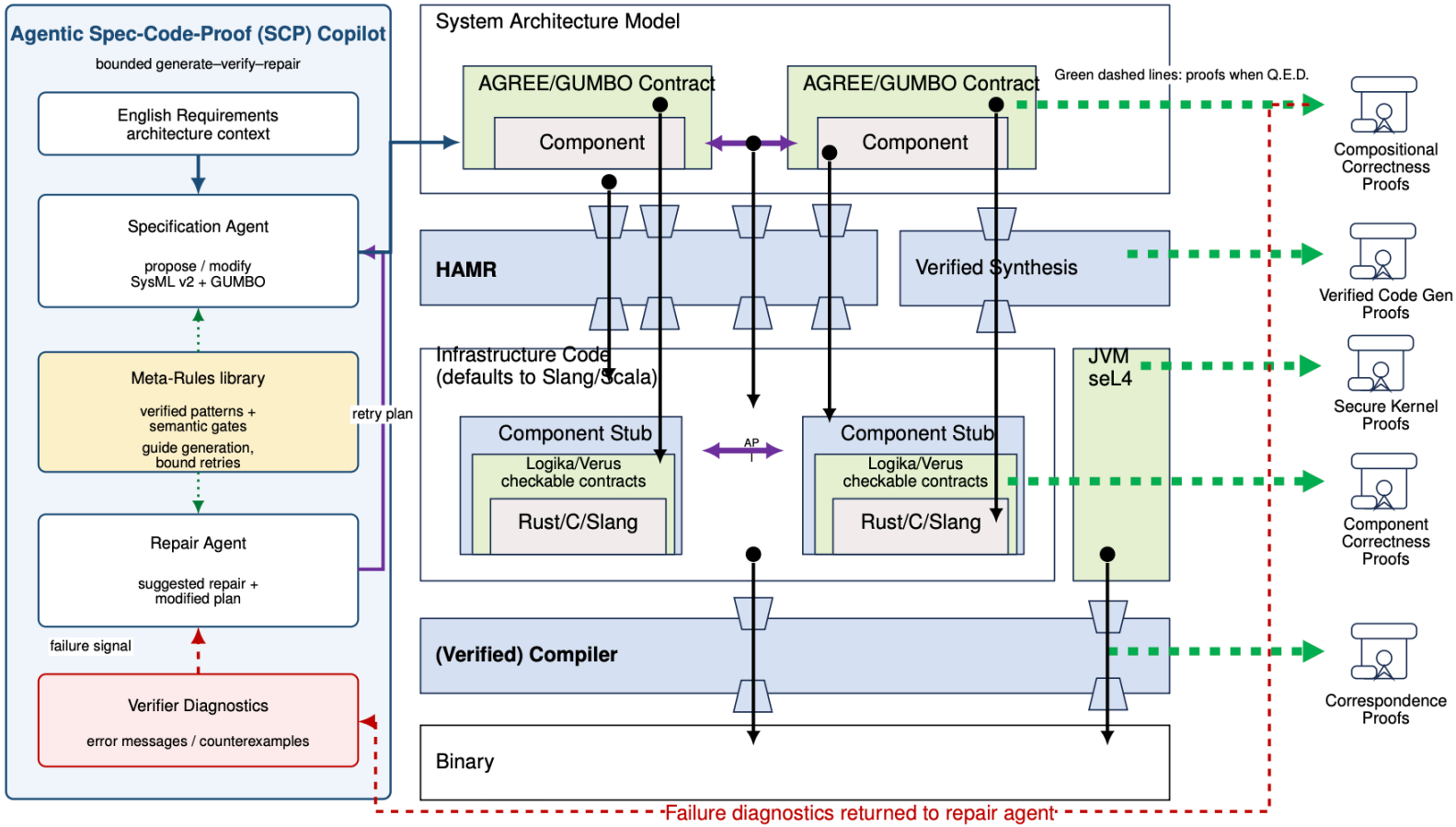
Spec-Code-Proof: *Formal* Specification-Driven Development

- ***Key Insight:*** Specification-Driven Development can be augmented to create *Formal* Specification-Driven Development
  - FSDD addresses an issue rarely discussed amongst all of the AI hype: how do we use generative AI to improve **quality and assurance**

- Brad Martin, Collins, and Stanford anticipated this general need more than two years ago:
  - Spec-Code-Proof (SCP) added task on PROVERS uses LLMs to assist in Model-Based Systems Engineering development, including SysML v2 models, Gumbo system contracts, and natural language requirements (Collins) as well as Rust code, Verus annotations, and natural language descriptions (Stanford)

- Our neuro-symbolic tools ensure (by automated SMT solving) that:
  - LLM-generated code implementations meet their (formal) specifications
  - LLM-generated specifications accurately reflect the implementation
  - Natural language requirements are faithfully translated to formal specs and implementations

INSPECTA Tools Plus Generative AI



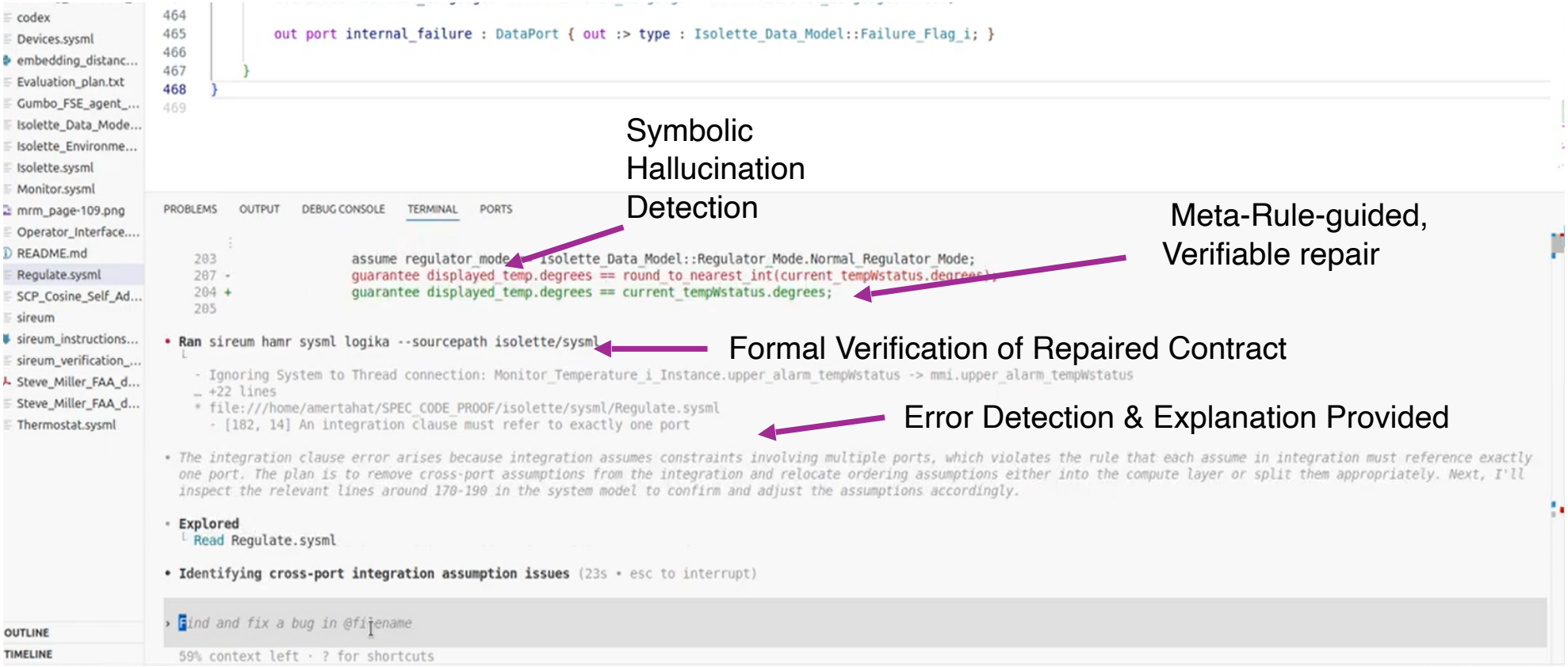
Example: Temperature Control System

- Requirements originate from an FAA Requirements Engineering Handbook
  - ~50 PDF pages devoted to the Temperature Controller
  - ~40 Requirements

- Challenge Problems: Given the Requirements, use an LLM to synthesize a SysML v2 model, and/or a set of Gumbo contracts for that model, and verify the contracts against the model using existing INSPECTA tools for SysML v2. Synthesize Rust/Verus code from the model using the existing HAMR tool.

- Results so far: Codex 5.5 generates ~40 Gumbo contracts that are successfully automatically verified by HAMR/Logika.
  - ~370K input tokens, ~52K output tokens, ~39 minutes wall-clock time

Gumbo Assume/Guarantee Contract Generation and Repair



Stanford Spec-Code-Proof Work

TACAS 2026

VeriStruct

Chuyue Sun[1], Yican Sun[2], Daneshvar Amrollahi[1], Ethan Zhang[1]
Shuvendu Lahiri[3], Shan Lu[3], David Dill[1], Clark Barrett[1]

[1]Stanford University • [2]Peking University • [3]Microsoft Research

The Verification Annotation Burden

The Annotation Burden

- Verification needs 2 expert artifacts: specs + proof hints
- Writing annotations is the #1 cost of formal verification adoption
- Middle/bottom-layer specs = pure proof infrastructure

What Verification Needs

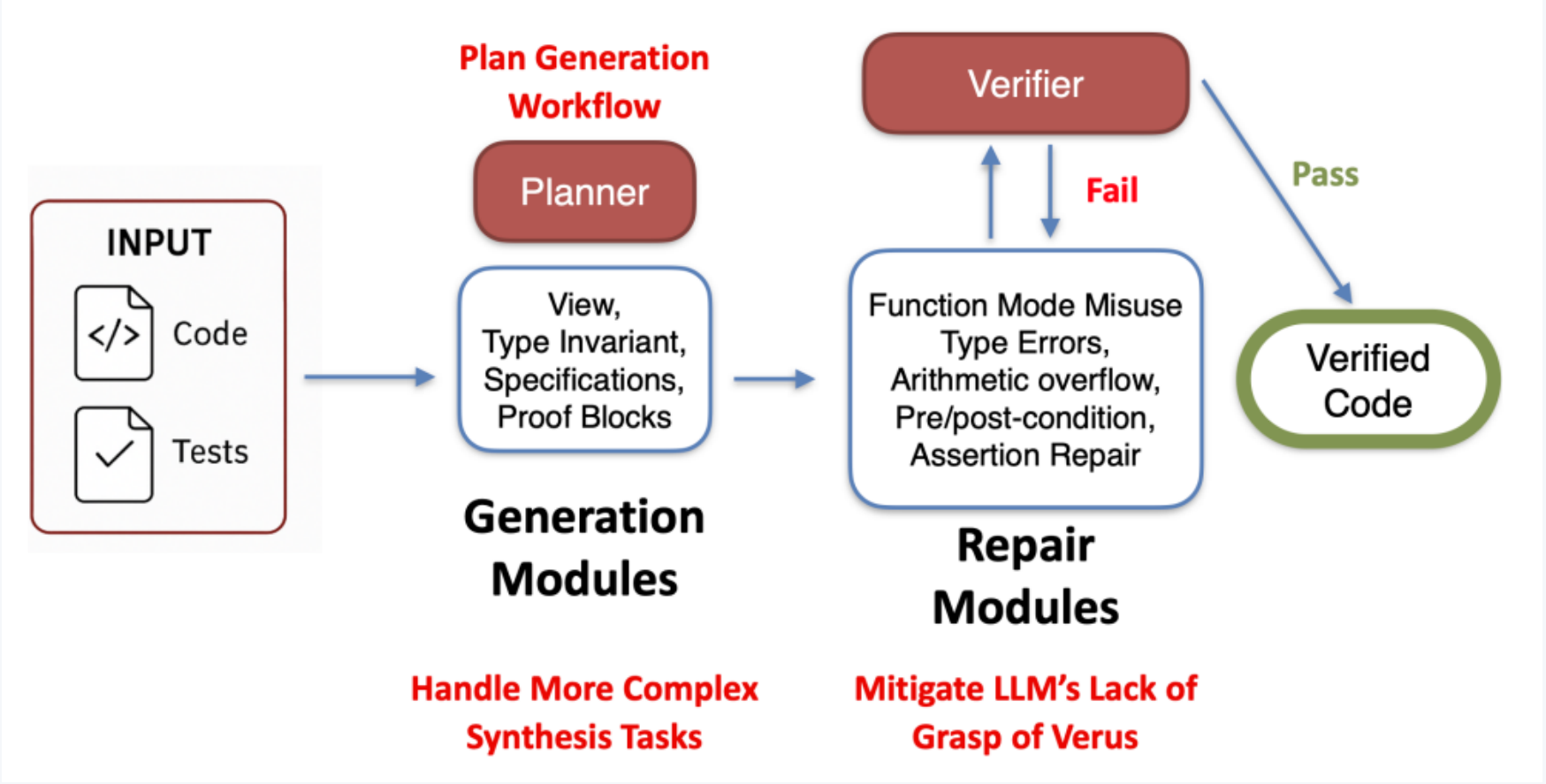
- **Specs:** requires / ensures
- **Proof hints:** invariants, assertions, lemmas
- Writing them is the primary cost of formal verification

Our Vision: AI-Generated Annotations, Human-Stated Intent

Humans state top-level contracts. Model infers all intermediate specs + proof hints, using the compiler as a sound verification oracle.

VeriStruct: first data-structure module verification

VeriStruct: Automated Verus Annotation and Proof for Rust Source



Targeted Repair Heuristics

Specialized modules route errors to targeted fixes — up to 5 repair rounds

Table 6: Specialized repair heuristics, the failure classes they target, and representative actions.

Heuristic	*Primary failures*	*Representative actions*
Syntax Repair	Parser and sequence syntax errors	Insert missing `view()` calls or lemmas, balance delimiters, and correct operators/generics.
Type Repair	Type mismatches and constructor invariant failures	Rewrite expressions, add explicit generics (e.g., `None::<T>`), and patch missing `requires` clauses.
Arithmetic Repair	Nonlinear reasoning and overflow/underflow	Emit `by (nonlinear_arith)` proofs, assert bounds, and strengthen loop invariants.
Precondition Repair	General preconditions, vector bounds, private access	Introduce proof blocks, assert length/index constraints, or adjust calls violating permissions.
Postcondition Repair	Failing `ensures` clauses	Add exit proofs, tune invariants, or expose ghost accessors for private fields.
Invariant Repair	Loop invariant failures pre- and post-loop	Assert invariants before loops, propagate conditions, or revise invalid invariants.
Missing Element Repair	Absent imports or trait implementations	Insert `use` statements and synthesize required trait methods with contracts.
Mode Repair	Mode mismatches and visibility issues	Wrap calls in spec/proof blocks, retag functions, or set open/-closed attributes.
Old-Self Repair	Missing `old(self)` refer-	Rewrite `self.` usages inside
Assertion Repair	Failed assertions and test expectations	Add reveals or lemmas (e.g., `lemma_seq_subrange_all`) and lift tests into postconditions.
Decrease Repair	Unsatisfied `decreases` obligations	Prove measure drops, adjust `decreases` expressions, or update loop variables.
Invariant Removal	Redundant invariant invocations	Delete unnecessary `self.inv()` calls when enforced by `#[verifier::type_invariant]`.

Key Takeaways

11 specialized repair modules each targeting a distinct Verus error class

Assertion Repair is unique: interprocedural analysis identifies the responsible method and strengthens its postcondition

Key design: spec repair is separate from proof repair

Most benchmarks converge within 3 rounds

Evaluation: Benchmarks

11 Rust data structure modules from Verus repository and open-source code

Benchmark	Description	#Functions
ATOMICS	An atomic counter in concurrent programming.	11
BITMAP	Bitmap implemented over 64-bit words.	14
TREEMAP	BST-based map with ordered key/value bindings.	21
INVARIANTS	Concurrent routines showing reusable invariants.	7
NODE	Implementing the node class of the binary search tree.	12
OPTION	Polymorphic option wrapper with safe accessor methods.	15
RINGBUFFER	Implementing a ring buffer.	13
RWLOCKVSTD	Read/write lock implementation.	5
SETFROMVEC	Set abstraction constructed from backing vectors.	10
TRANSFER	Account transfer routines that enforce balanced updates.	5
VECTORS	Implementing a vector with basic algorithms.	16

Total: 129 functions across diverse data structure types

Results: Outstanding Success Rate

128/129

Functions Verified

10/11

Benchmarks Solved

99.2%

Success Rate

Only failure: 1 function in Node benchmark (11/12 verified)

Comparison with Baselines

Method	Functions	Benchmarks	Tokens/bench
Single-shot baseline	52	4/11	—
Claude Code (Sonnet 4.5)	102	8/11	~24k
VeriStruct (ours)	**128**	**10/11**	**~22k**

Model: o1 (Dec 2024) • n=3 • m=5 repair rounds | Note: current SOTA models (e.g. Opus 4.6) far exceed o1's capabilities and can easily solve all benchmarks

Future Work: A Lean-Native Verification Pipeline

CSLib's Boole IVL + Lean backends as the natural successor to Verus + Z3

VeriStruct today

```
Rust + Verus annotations
↓
Verus IVL (Rust subset + ghost state)
↓
Z3 (SMT)
```

Limitations

- Nondeterministic search: identical code can flake (dalek-lite: 40+ revisions)
- Undecidable on nonlinear arithmetic; weak on induction
- AI provers (AlphaProof, DSP-V2) target Lean, not SMT

CSLib direction

```
Rust / C++ / Python + Lean specs
↓
Boole IVL  (CSLib pseudocode IVL)
↓
Lean VCs → Grind, Hammer, AI provers
```

What this unlocks

- Deterministic: Lean kernel is small, heuristic-free
- Specs use full Mathlib + CSLib abstractions
- AI-prover-native; smaller TCB

VeriStruct's structured workflow ports cleanly: swap the IVL backend, keep the planner / generator / repair cascade.

Trend 5:

Formal Verification for Generative AI

Synthesis using Generative AI: The Good

- The ability of Generative AI to process natural language is a major step forward, as most systems requirements are expressed informally in natural language
- The ability of LLMs to generate source code from higher-level specifications, and do so quickly, has improved significantly over the past few years
- The recent emphasis on specification-driven development is welcome (but writing good specs is still very difficult)
- LLMs are able to handle the “long tail” of computer science languages surprisingly well

Synthesis using Generative AI: The Bad

- Hallucinations are still a significant issue
- Basic arithmetic processing is still poor
- Weakness in access control has allowed LLMs to pass a test suite by deleting the test cases, or deleting a company's entire production database without asking (PocketOS)
- Agents skipping steps in development workflows
- Jailbreaks
- Agent harness vulnerabilities/exploits (e.g., recent PyTorch Lightning supply chain attack)
- LLMs are much better at generating code than refining or debugging it
- LLMs are much too "eager to please" and thus are not very useful in software inspection roles

Synthesis using Generative AI: The Ugly

- Agentic frameworks are becoming a large “wad” of unverified, lightly tested (mainly) Python code that is given too much access to project artifacts
  - Generative AI developers generally lack expertise in secure software development
  - Developers accustomed to a “move fast and break things” culture have little appreciation for the “take care and get it right” critical systems development philosophy
- Users think that LLM-generated code is more secure than hand-written code, when in fact it is not
- Generative AI developers toss around terms like “inference” and “chain of thought” when their tools merely “cosplay” at reasoning
- Markdown files are a poor way to capture specifications/developer intent

Fixing the “Bad” and “Ugly”

- In order for Generative AI to be utilized in high-assurance, critical applications, we must address the “bad” and “ugly” shortcomings

- Under the DARPA PROVERS program, the Collins Aerospace team is currently developing a proof-of-concept for RASE, a Rigorous Agentic Systems Engineering framework that takes on many of these issues
  - The first RASE demos are slated for the next PROVERS PI meeting, to be held at Stanford at the end of September

Rigorous Agentic Systems Engineering (RASE) Goals

- Thwart hallucinations via neuro-symbolic techniques
  - e.g., Spec-Code-Proof, Assume/Guarantee contracts, programming language annotations
- Elevate GenAI Specification-Driven Development by supporting standardized Systems Engineering languages (e.g., SysML v2) and annotations, as well as development language annotations, that can be mathematically analyzed
  - Not just unstructured Markdown files
- Specify Workflows as well as Agent policies and roles to support rigorous Systems Engineering with strong Guardrails
  - Detail which artifacts agents can read/write, which tools they can access, etc.
- Implement agents using verification-enhanced programming languages, enabling formal verification of agentic behavior
- Provision rigorous, OS-enforced Sandboxing

What Does All This Mean for the seL4 Community?

- As pioneers in the area of Formal Specification-Driven Development, we can both take advantage of this upswing in FV mindshare, as well as serve as thought leaders for future critical systems development
  - One of the biggest “I told you so” moments in computing history :-)

- Our DARPA PROVERS work has demonstrated that recent advances in FV have made the creation of verified seL4 applications much easier for non-specialists
  - Our PROVERS work has also led to a significant increase in the number of supported seL4 platforms, thanks in no small part to proof engineering efforts

- As Generative AI agents escape their sandboxes and are subjected to increased Government scrutiny, the “frontier labs” will, by necessity, become cognizant of high-assurance development tools, techniques, and design rules, including seL4