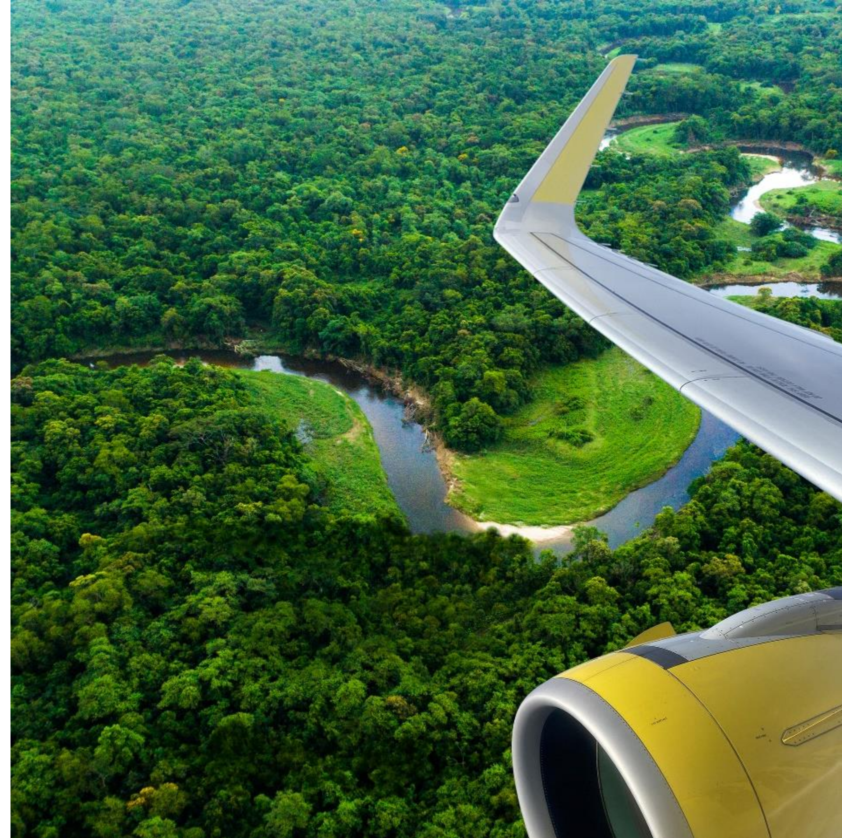


PROVERS

Collins Aerospace
An RTX Business

High-Assurance Synthesis and Analysis Techniques for Memory-Safe Programming Languages

David S. Hardin, Ph.D.
Chief Technologist, Trusted Methods
Applied Research and Technology

© 2024 Collins Aerospace.
All rights reserved.

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).

This document does not include any export-controlled technical data.

Disclaimer

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

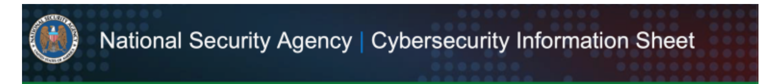
Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

This document does not include any export-controlled technical data.

Motivation

- An emerging consensus amongst computer science thought leaders is that memory-safe programming language technology needs to be adopted more broadly:

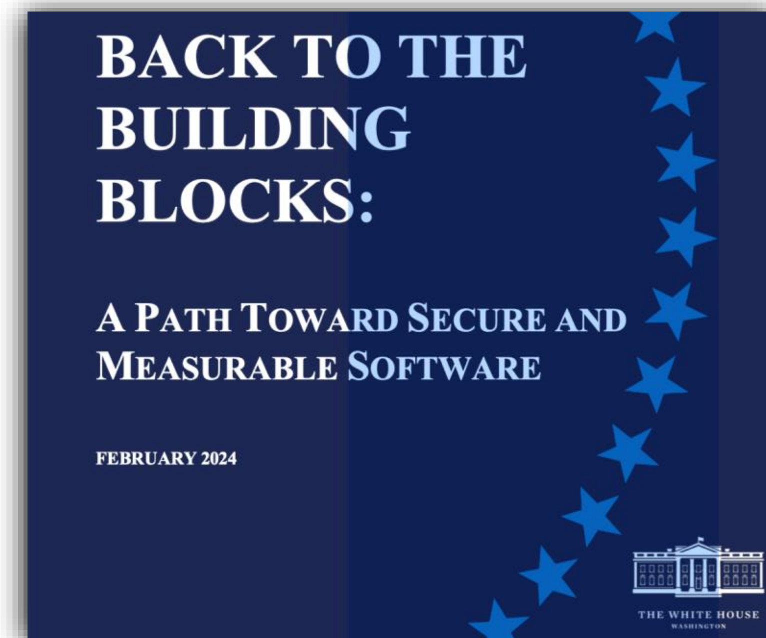
- “NSA recommends using a memory safe language when possible.” (11/2022)
- The White House has published a report championing the adoption of memory safe programming languages to enhance software security. (02/2024)
- Microsoft, Google, and Amazon have all announced significant Rust initiatives.
- A Rust development environment has achieved ISO 26262 and IEC 61508 certification



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



Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Why Memory-Safe Languages? Why now?

- Memory-safe languages are not new
  - Collins successfully used Ada in major commercial and government avionics products in the 1980s and 1990s
  - 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, leading to a never-ending parade of security vulnerabilities, despite the use of increasingly sophisticated analysis tools

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Memory-Safe Languages: Synthesis and Analysis

- In the rest of this talk, I will focus on examples of how the Trusted Methods team and our research partners are currently moving aggressively to support memory-safe languages in

- Formal, Automated Synthesis, and
- Formal, Automated Analysis

- of high-assurance systems

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export-controlled technical data.

Synthesis

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Memory-Safe Language Synthesis Technologies

- Given the advantages of memory-safe languages cited in the NSA and White House reports, it seems natural to target memory-safe languages for software synthesis tasks

- We are pursuing a number of formal synthesis scenarios, including:
  - Code synthesis from a Model-Based Systems Engineering model
  - High-assurance source-to-source transpilation
  - Code synthesis from a formal specification

- NB: The seL4 Foundation is supporting memory-safe language synthesis for a verified operating system environment by funding the development of Rust bindings for seL4
  – Full Disclosure: The author is on the Board of Governors of the seL4 Foundation

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).

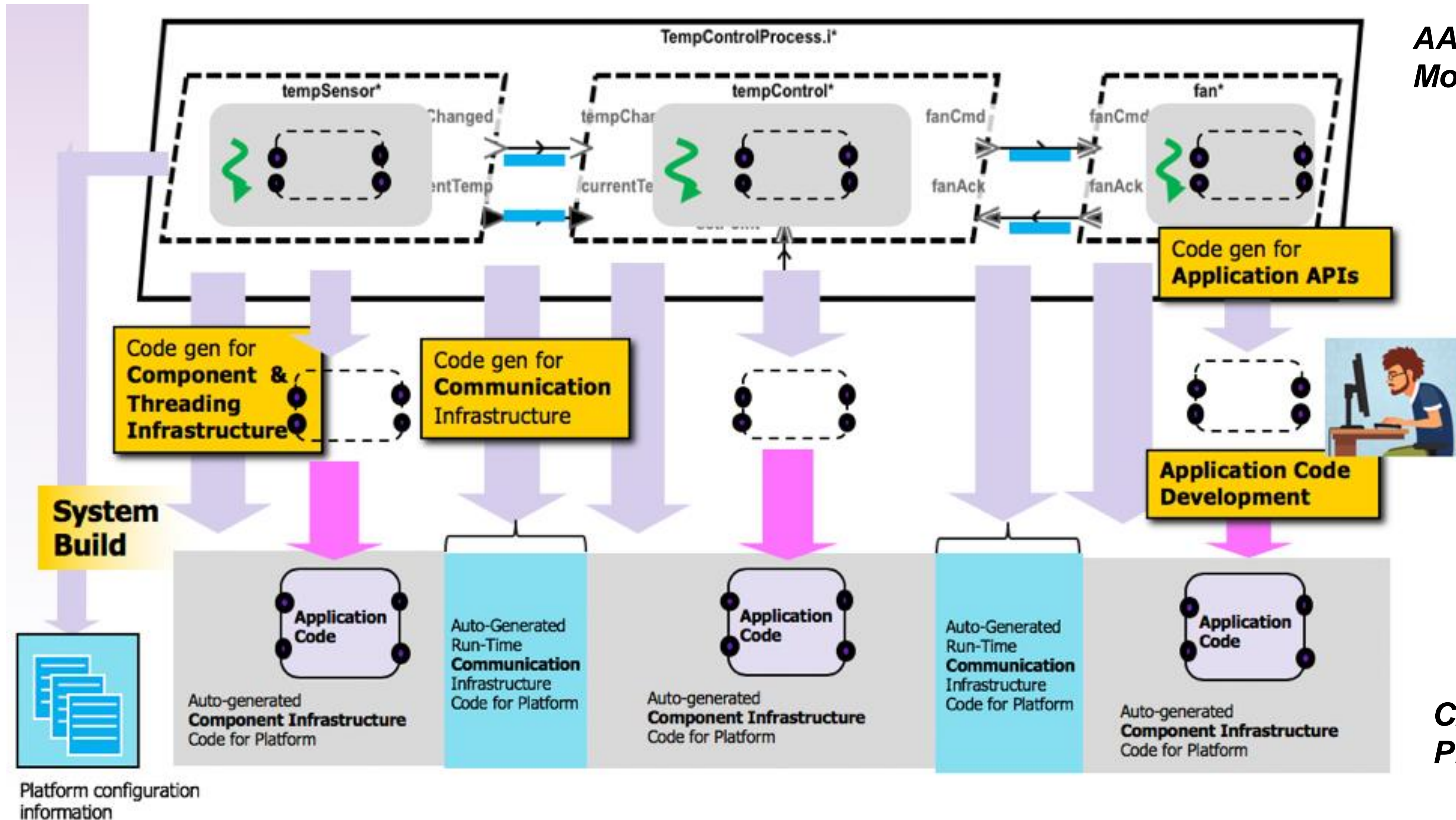
This document does not include any export-controlled technical data.

Kansas State University: Code Synthesis for Model-Based Systems Engineering with HAMR

- HAMR generates skeletal implementation code from architecture models in either AADL or SysML v2
- HAMR translates architecture models into Slang, a subset of Scala
  - The Kansas State team has developed an SMT-based formal analysis capability for Slang, called Logika
- KSU developed a transpiler from Slang to “Embedded C” for the DARPA CASE program
  - With this capability, HAMR can target seL4, Linux, or a JVM-based simulator with a flick of a switch
- For DARPA PROVERS, KSU is developing a Slang-to-Rust transpiler

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

HAMR Overview



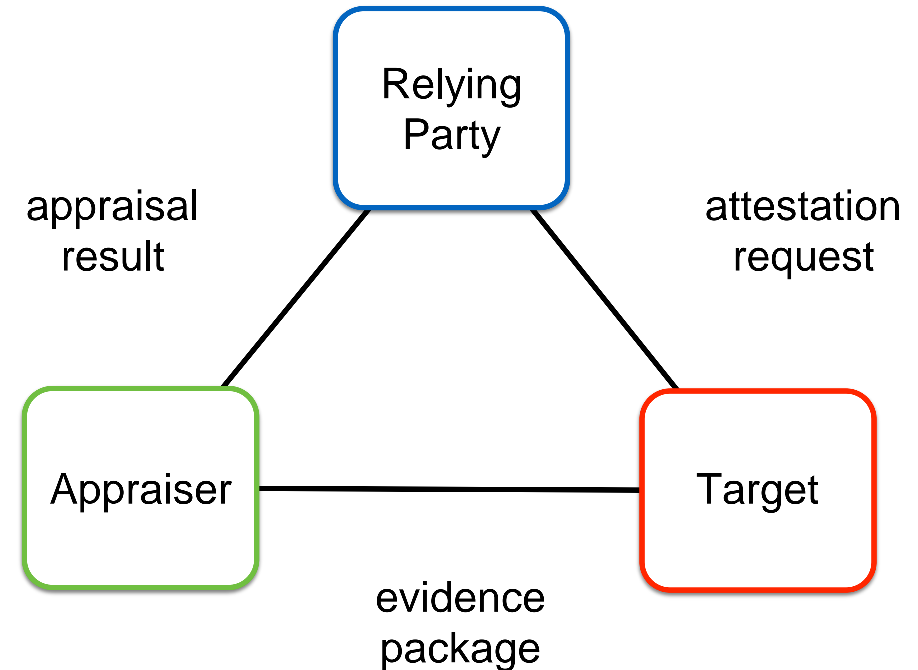
AADL/SysML v2 Model

CASE: C
PROVERS: Rust

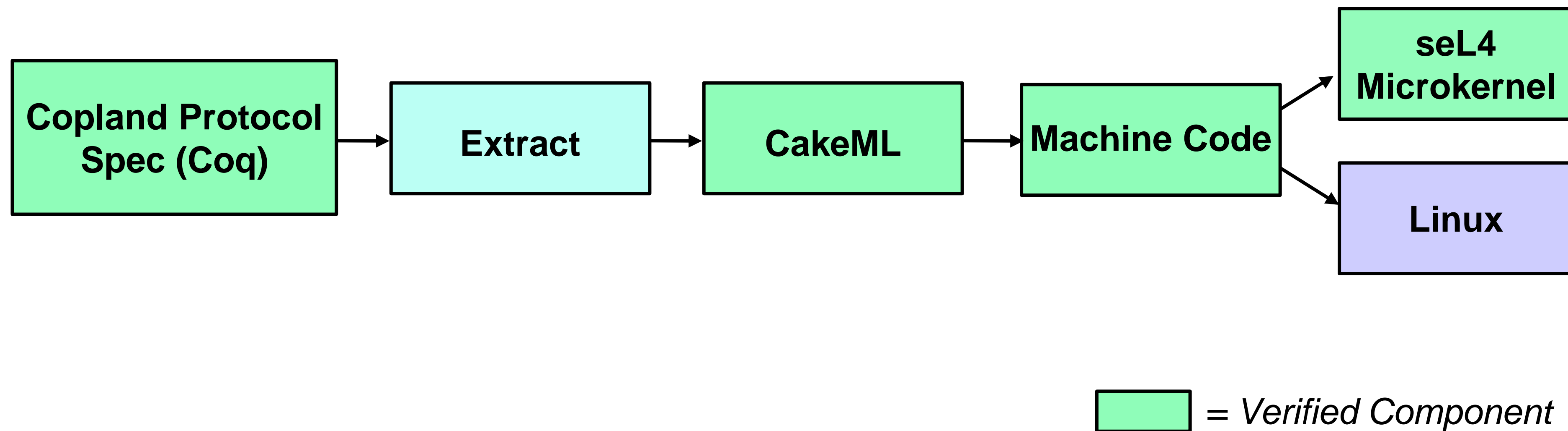
Collins Aerospace An RTX Business
K·STATE
Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Copland Verified Remote Attestation (University of Kansas)

- Measurement and Attestation
  - gathering evidence of booting system
  - gathering evidence of executing system
  - gathering evidence of evidence gathering
- Appraisal
  - evaluating evidence of expectation
  - is a system behaving as expected?
- Today - Boot and runtime appraisal
  - relying party requires trust
  - attestation generates evidence
  - appraiser checks expectations over evidence
- Tomorrow - Systems over time
  - records and ledgers for evidence
  - system and local manifests for configuration
  - flexible mechanism for system appraisals



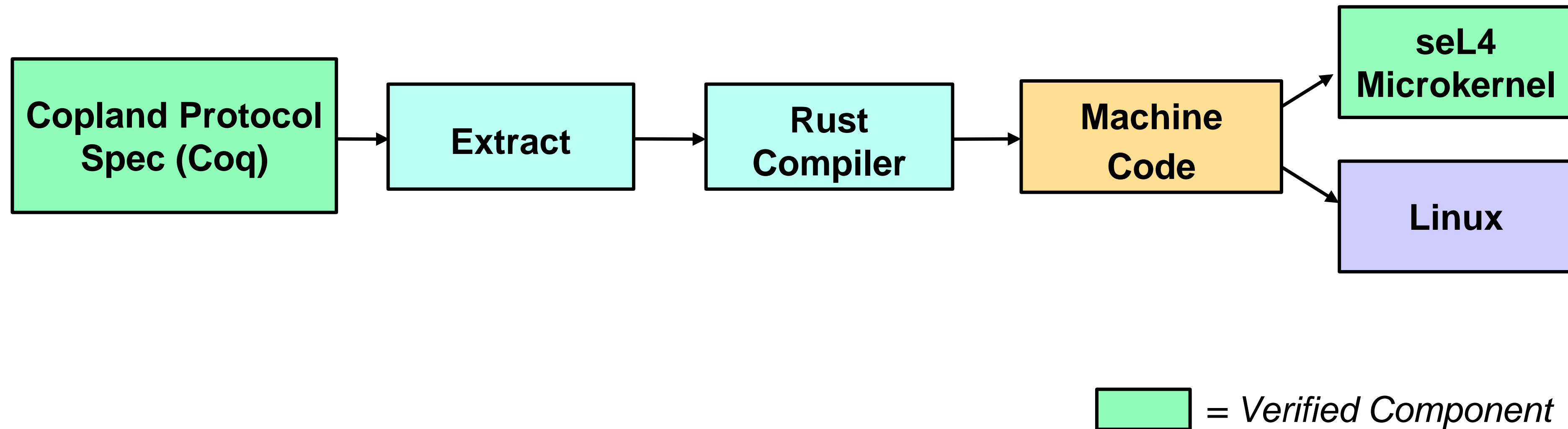
KU Copland Remote Attestation Protocol Synthesis (DARPA CASE)



Remote Attestation Protocol Implementation Synthesized from Coq Spec

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

KU Copland Remote Attestation Protocol Synthesis (Plan for DARPA PROVERS)



Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Analysis

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Verification Tools for Memory-Safe Languages

- A number of verification tools have been developed for Rust, including:
  - Cruesot (Inria)
  - Prusti (ETH Zurich)
  - RustHorn (University of Tokyo/Chiba University)
  - Kani (Amazon)
  - Verus (Carnegie-Mellon University)

- With Verus (see 2024 HCSS talk for details), developers express proofs and specifications using Rust syntax, allowing proofs to take advantage of Rust’s linear types and borrow checking.
  - Note: The Verus team is part of the Collins DARPA PROVERS effort.

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Verification Tools for Memory-Safe Languages (cont'd.)

- AdaCore provides a verification toolsuite, GNATprove, for SPARK 2014
  - SPARK guarantees a number of important program properties, including exception freedom, array indices stay in bounds, etc.
  - SPARK also provides language support for contracts, with preconditions, postconditions, data dependencies, control dependencies, termination guarantees, etc.
  - GNATprove frontends modern SMT solvers, such as Z3 and CVC4

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Hardware/Software Co-Design and Co-Assurance

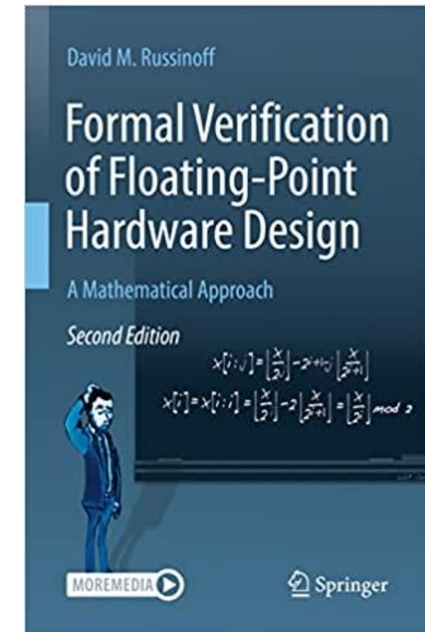
- We desire to create high-assurance components using hardware/software co-design/co-assurance techniques
- The high-level Architecture Models in CASE, and now PROVERS supports both hardware- and software-based realizations
- The ability to defer and/or change the allocation of functionality to hardware or software provides development flexibility
- Hardware provides greater tamper resistance, as well as higher performance

- Hardware/Software Co-Design is enabled by a High-Level Specification Language (HLS), which is closer to mainstream programming languages than is a Hardware Description Language such as VHDL or Verilog
  - Most HLS's are based on C
  - However, memory-safe languages are a natural match for hardware
  - Thus, we see a need for Hardware/Software Co-Assurance using a memory-safe programming language such as Rust

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Hardware/Software Co-Assurance using Rust

- The hardware/software verification approach we leverage was developed by David Russinoff and colleagues
- Their approach is called Restricted Algorithmic C (RAC), as it is based on Mentor's HLS Algorithmic C
  - RAC is extensively documented in Russinoff's book, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*
    - In Russinoff's text, RAC is applied to the verification of realistic Arm floating-point designs using the ACL2 theorem prover
    - RAC, and the verifications described in the book, are all available in the standard ACL2 theorem prover distribution



Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

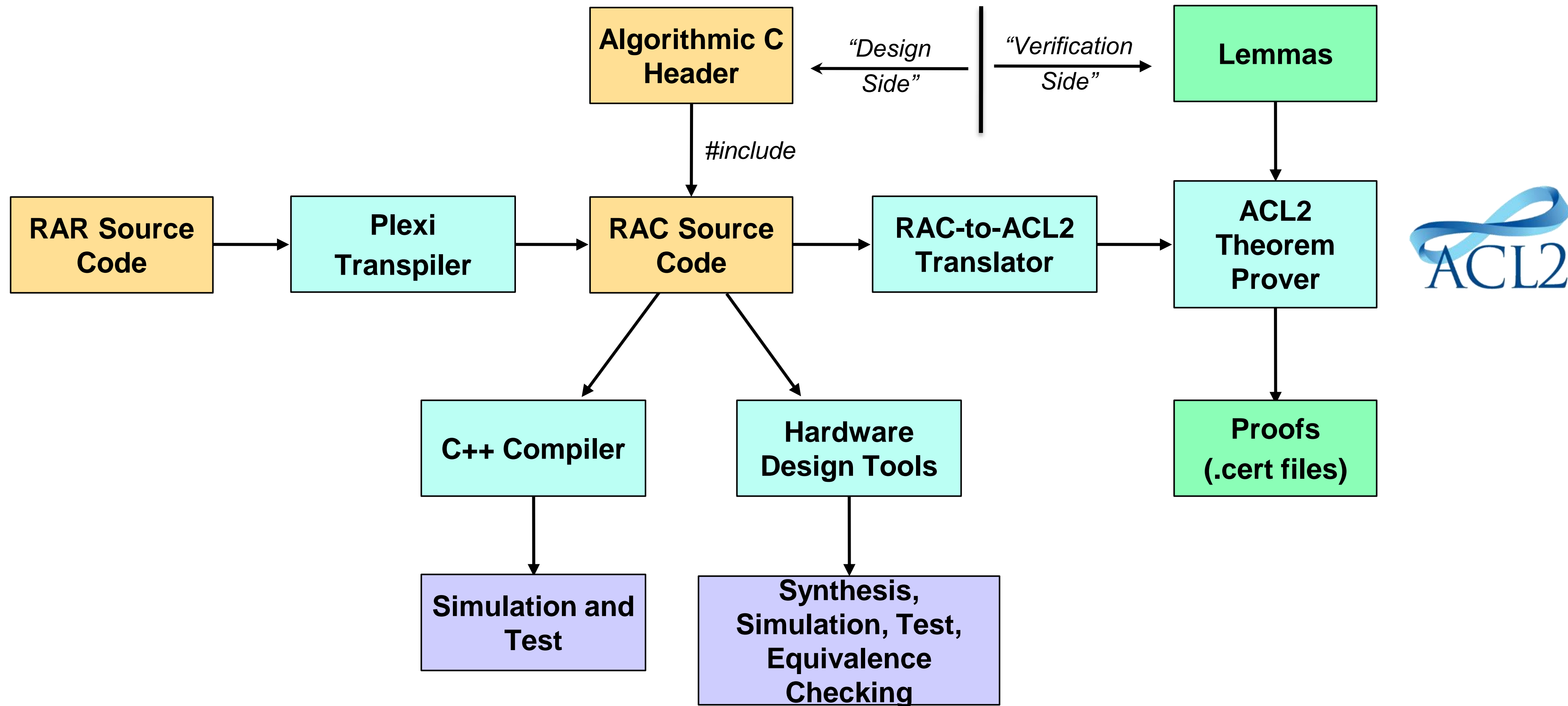
Restricted Algorithmic Rust (RAR)

- RAR currently is a simple Rust frontend to the RAC toolchain
  - RAR has RAC semantics, with Rust syntax

- We have created a number of examples using the RAR toolchain, including:
  - Array-Backed Verified Algebraic Data Types: Stack, Singly-linked list, Doubly-linked list, Circular Queue, Deque, etc.
  - A DFA-based JSON lexer, coupled with an LL(1) JSON parser
  - A significant subset of the Monocypher modern cryptography suite

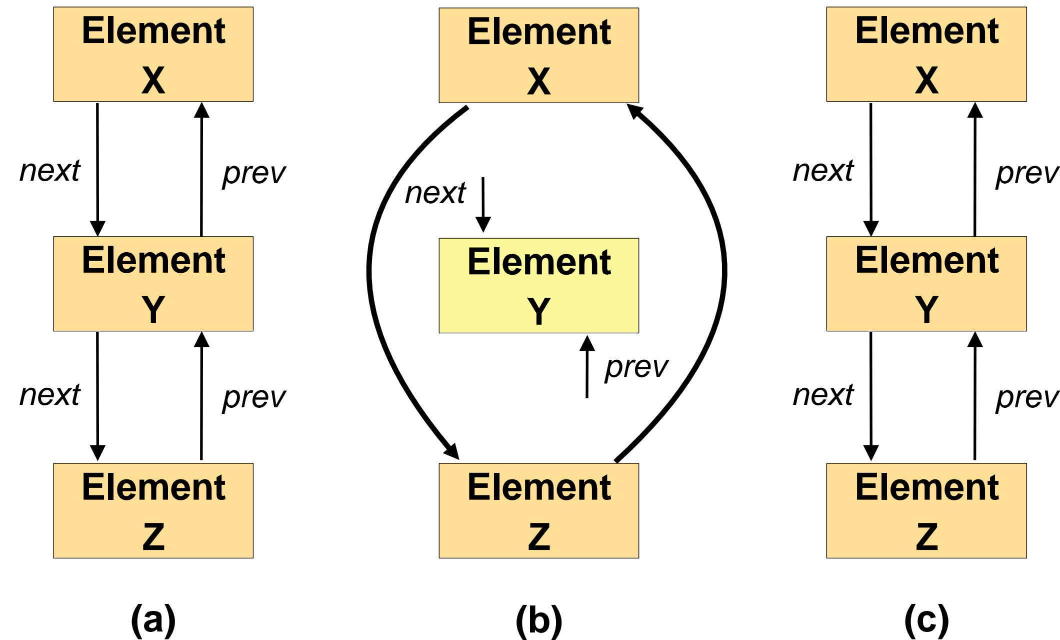
Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

RAR Toolchain



Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Example: Knuth's "Dancing Links" for Exact Cover Problems



TAOCP, vol 4B (2022)

(a) Doubly-linked circular list portion prior to remove operation.

(b) After remove of element Y.

(c) After restore of element Y.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Dancing Links in RAR

- A circular doubly-linked list (CDLL) is specified in RAR as follows:

```
const CDLL_MAX_NODE1: usize = 8191;   // arbitrary
const CDLL_MAX_NODE: usize = CDLL_MAX_NODE1 - 1;

#[derive(Copy, Clone)]
struct CDLLNode {
  alloc: u2,
  val: i64,
  prev: usize,
  next: usize,
}

#[derive(Copy, Clone)]
struct CDLL {
  nodeHd: usize,
  nodeCount: usize,
  nodeArr: [CDLLNode; CDLL_MAX_NODE1],
}
```


Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Dancing Links remove()

```
fn CDLL_remove(n: usize, mut CDObj: CDLL) -> CDLL {
  if (n > CDLL_MAX_NODE) {
    return CDObj;
  } else {
    if (n == CDObj.nodeHd) {  // Can't remove head
      return CDObj;
    } else {
      if (CDObj.nodeCount < 3) {  // Need three elements
        return CDObj;
      } else {
        let nextNode: usize = CDObj.nodeArr[n].next;
        let prevNode: usize = CDObj.nodeArr[n].prev;

        CDObj.nodeArr[prevNode].next = nextNode;
        CDObj.nodeArr[nextNode].prev = prevNode;

        CDObj.nodeCount = CDObj.nodeCount - 1;

        return CDObj; } } } }
```


Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Translation to ACL2

```
(DEFUND CDLL_REMOVE (N CDOBJ)
     (IF1 (LOG> N (CDLL_MAX_NODE))
          CDOBJ
          (IF1 (LOG= N (AG 'NODEHD CDOBJ))
               CDOBJ
               (IF1 (LOG< (AG 'NODECOUNT CDOBJ) 3)
                    CDOBJ
                    (LET* ((NEXTNODE (AG 'NEXT (AG N (AG 'NODEARR CDOBJ))))
                           (PREVNODE (AG 'PREV (AG N (AG 'NODEARR CDOBJ))))
                           (CDOBJ (AS 'NODEARR
                                      (AS PREVNODE
                                          (AS 'NEXT
                                              NEXTNODE
                                              (AG PREVNODE (AG 'NODEARR CDOBJ)))
                                          (AG 'NODEARR CDOBJ))
                                      CDOBJ))
                           (CDOBJ (AS 'NODEARR
                                      (AS NEXTNODE
                                          (AS 'PREV
                                              PREVNODE
                                              (AG NEXTNODE (AG 'NODEARR CDOBJ)))
                                          (AG 'NODEARR CDOBJ))
                                      CDOBJ)))
                          (AS 'NODECOUNT
                              (- (AG 'NODECOUNT CDOBJ) 1)
                              CDOBJ))))))
```

Collins Aerospace
An RTX Business

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).

This document does not include any export controlled technical data.

Dancing Links Correctness

```
(defthm restore-of-remove--thm
 (implies
  (and (cdllp Obj)
       (good-nodep n Obj)    ;; various well-formedness predicates
       (not (= n (ag 'nodeHd Obj)))
       (>= (ag 'nodeCount Obj) 3))
  (= (CDLL_restore n (CDLL_remove n Obj))
     Obj)))
```

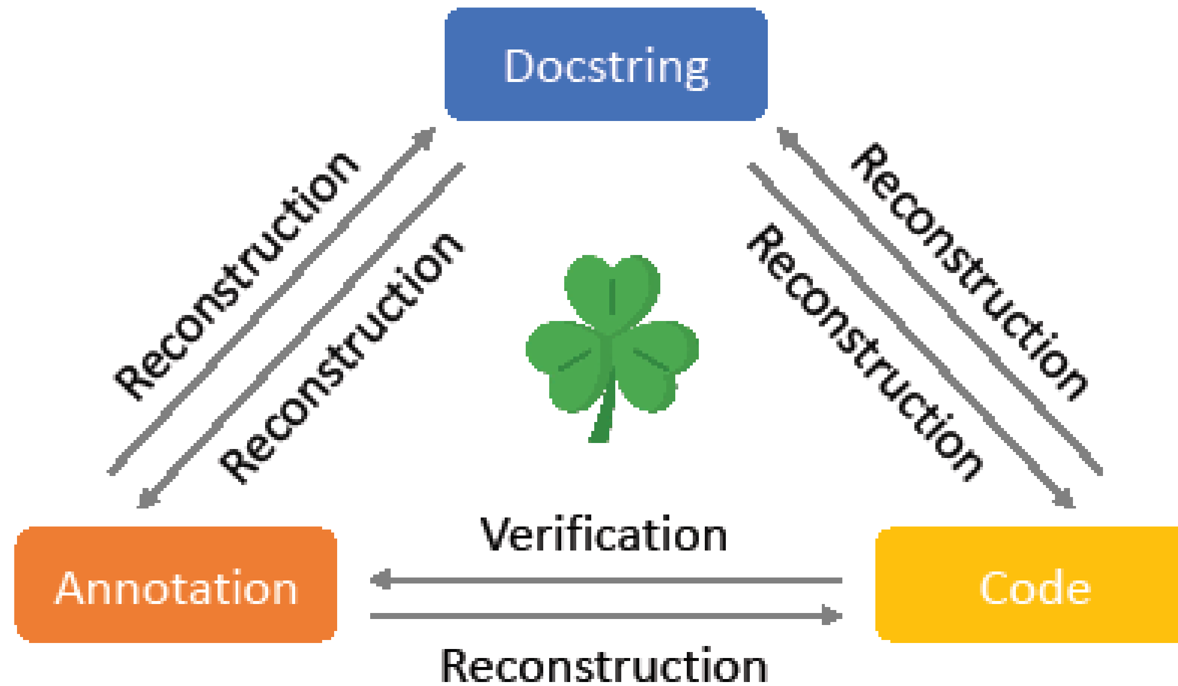
ACL2 proves 160 circular doubly-linked list functional correctness lemmas and theorems completely automatically.

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

One More Thing…
Synthesis + Analysis

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Clover: Generative AI for Memory-Safe Language Specs, Code, and Natural Language Descriptions



…stay tuned…

Conclusion and Recommendations (IMHO)

- Modern Memory-Safe Languages can make significant contributions to improved software assurance, with little to no degradation in performance
  - Thus, formal tool providers should support memory-safe language synthesis
- In general, memory-safe languages are much easier to reason about
  - Thus, formal methods researchers should aggressively support memory-safe languages for automated formal analysis
- Academia should teach these languages, explain their advantages, and disabuse students of the notion that bug-filled code is inevitable
- Industry should "skate to where the puck is going to be" and begin adoption plans for memory-safe languages in their products *now*
- Government Officials should stand behind their stated intent to move to memory-safe languages, and help industry and academia to realize that vision

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

This document does not include any export controlled technical data.

Acknowledgments

- Thanks to all our partners mentioned:

- This work was funded in part by DARPA

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

This document does not include any export controlled technical data.

Thanks

- Questions?

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited).
This document does not include any export controlled technical data.

Collins Aerospace
An RTX Business