Technical Volume

NRC, Solicitation No. 31310021R0046

High Assurance Rigorous Digital Engineering for Nuclear Safety

Contents

List of Figures

List of Tables

Glossary

Coq Coq is an interactive theorem prover first released in 1989. It allows for expressing mathematical assertions, mechanically checks proofs of these assertions, helps find formal proofs, and extracts a certified program from the constructive proof of its formal specification. Coq works within the theory of the calculus of inductive constructions, a derivative of the calculus of constructions. Coq is not an automated theorem prover but includes automatic theorem proving tactics (procedures) and various decision procedures. 4, 8, 12, 23, 29

Cryptol Cryptol is a domain specific programming language for cryptography developed by Galois. The language was originally developed for use by the United States National Security Agency. The language is also used by private firms that provide information technology systems, such as Amazon and defense contractors in the United States. The programming language is used for all aspects of developing and using cryptography, such as the design and implementation of new ciphers and the verification of existing cryptographic algorithms. 2–4, 7, 8, 10–13, 16, 19, 20, 22, 23, 30, 31

DevSecOps The use of tools in a user's local and remote design, development, validation, verification, maintenance, and evolution environments that facilitate the automatic and continuous evaluation by static and dynamic means of a system/subsystem/component's behavioral (e.g., safety and correctness) and non-behavioral (e.g., well-formedness and security) properties. 11

Digital Instrumentation and Control Systems One of several types of control systems and associated instrumentation used for industrial process control. Such systems can range in size from a few modular panel-mounted controllers to large interconnected and interactive distributed control systems with many thousands of field connections. Systems receive data from remote sensors measuring process variables (PVs), compare the collected data with desired setpoints (SPs), and derive command functions which are used to control a process through the final control elements (FCEs), such as control valves. 1, 5, 13, 22, 24

FRET The NASA Formal Requirements Elicitation Tool is used to make writing, understanding, and debugging formal requirements natural and intuitive. 4, 12, 19

PVS A specification language integrated with support tools and an automated theorem prover, developed at the Computer Science Laboratory of SRI International. PVS is based on a kernel consisting of an extension of Church's theory of types with dependent types, and is fundamentally a classical typed higher-order logic. 4, 8, 12

Requirements State Modeling Language A formal specification language that uses hierarchical finite state machines to specify system requirements. 12, 19

RISC-V RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use. A number of companies are offering or have announced RISC-V hardware, open source operating systems with RISC-V support are available and the instruction set is supported in several popular software toolchains. 1, 6, 9–11, 15–17, 19, 20, 23, 24, 26, 30

SAT The Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. 23

SAWscript The proof script language is used to specify the assumptions and proof goals of formal verifications to the SAW tool. 8, 12, 23

soft-core A CPU or SoC that is implemented in an HDL and synthesized to a bitstream and loaded onto an FPGA. 11, 15, 26

SPARK A formally defined computer programming language based on the Ada programming language, intended for the development of high integrity software used in systems where predictable and highly reliable operation is essential. It facilitates the development of applications that demand safety, security, or business integrity. 4, 10, 12, 24

SpeAR An integrated development environment for formally specifying and rigorously analyzing requirements. 12, 19

Verified Software Toolchain A software toolchain that includes static analyzers to check assertions about a C program; optimizing compilers to translate a C program to machine language; and operating systems and libraries to supply context for the C program. The Verified Software Toolchain project assures with machine-checked proofs that the assertions claimed at the top of the toolchain really hold in the machine-language program, running in the operating-system context. 12

Verifier for Concurrent C VCC is a program verification tool that proves correctness of annotated concurrent C programs or finds problems in them. VCC extends C with design by contract features, like pre- and postcondition as well as type invariants. Annotated programs are translated to logical formulas using the Boogie tool, which passes them to an automated SMT solver Z3 to check their validity. 12

Acronyms

21CC 21st Century Cryptography. 11, 30

AADL Architecture Analysis and Design Language. 4, 10, 12

ACSL ISO ANSI C Specification Language. 4, 12, 22, 23

AES Advanced Encryption Standard. 11, 12, 30

API Application Programming Interface. 29

ASIC Application-Specific Integrated Circuit. 7, 11, 24, 30

ASM Abstract State Machine. 12, 17

AWS Amazon Web Services. 8, 11, 13, 31

BESSPIN Balancing the Evaluation of System Security Properties Against Industrial Needs. 11

BLESS Behavioral Language for Embedded Systems with Software. 10

BON Business Object Notation. 12

BSV Bluespec SystemVerilog. 4, 8, 9, 11, 12, 16, 18, 19, 23, 24

BTS BESSPIN Tool Suite. 3, 12

CD Continuous Deployment. 8, 18

CI Continuous Integration. 5, 8, 11, 18

COTS Commercial Off The Shelf. 7–10, 12, 15, 26, 29

CPU Central Processing Unit. 1, 3, 11, 15–17, 23, 24, 26, 29

CSP Communicating Sequential Processes. 13

CV Continuous Verification. 5, 8, 11, 18

DE Digital Engineering. 11

DIB Defense Industrial Base. 2

DoD Department of Defense. 2, 3, 5, 13, 31

DSL Domain Specific Language. 12

EDA Electronic Design Automation. 2, 8

FPGA Field Programmable Gate Array. 7, 11, 15–17, 19, 20, 23, 24, 26

GCC Gnu Compiler Collection. 10

GFE Government Furnished Equipment. 18

GLASS-CV Galois Low-energy Asynchronous Secure SoC for Computer Vision. 30

GPIO General Purpose I/O. 15, 17, 26

HDL Hardware Description Language. 4, 11, 16

HMAC Hash-based Message Authentication Code. 11, 30, 31

HOL Higher-Order Logic. 4, 22–24

HSM Hardware Security Module. 30

IC Intelligence Community. 3, 5, 13, 30

IDE Integrated Development Environment. 6

IP Intellectual Property. 10, 29

IR Intermediate Representation. 9

ISA Instruction Set Architecture. 3, 6, 15, 23, 29

JML Java Modeling Language. 4, 12

LLVM Low Level Virtual Machine. 9, 10, 12

MBE Model-Based Engineering. 10, 11, 13, 24

MBSE Model-Based Systems Engineering. 1, 3, 11

NLP Natural Language Processing. 3

NPP Nuclear Power Plant. 1, 3, 5, 13, 20, 24, 25

NRC Nuclear Regulatory Commission. 1, 3, 5, 7, 8, 10, 11, 13, 17–19, 22, 24–26

NSA National Security Agency. 1, 2, 7, 8, 11, 13

OCL Object Constraint Language. 10

OSA Open Systems Architecture. 11

PPAS Power, Performance, Area, and Security. 29

RDE Rigorous Digital Engineering. 1, 2, 4, 6, 7, 11–13, 18, 22, 24–26, 29

RTL Register Transfer Level. 11

RTS Reactor Trip System. 13–21

SAW Software Analysis Workbench. 7–9, 11, 12, 23, 31

SCADE Safety Critical Application Development Environment. 10

SHA Secure Hash Algorithm. 11, 30

SMT Satisfiability Modulo Theories. 22, 23

SNFM Secret Ninja Formal Methods. 5, 11

SoC System-on-Chip. 3, 10, 11, 16, 24, 29

SoW Statement of Work. 18, 21

SSITH System Security Integration Through Hardware and Firmware. 11, 18, 19, 29

SV SystemVerilog. 8, 9, 11, 12, 16, 18, 19, 23, 24, 30

SVA SystemVerilog Assertions. 4, 11, 22, 23

SWaP Size, Weight, and Power. 29

SysML System Modeling Language. 4, 10, 12, 19

UI User Interface. 15, 17, 26

UML Unified Modeling Language. 10, 12

USB Universal Serial Bus. 15

USG United States Government. 2

UTP Unified Theories of Programming. 12

UVM Universal Verification Methodology. 4, 8, 23

UX User eXperience. 26

VDM Vienna Development Method. 8, 12

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. 7

1. Introduction

1.1. Project Goals

Galois is pleased to propose to the Nuclear Regulatory Commission (NRC) the following Statement of Work for the **H**igh-**A**ssurance **R**igorous **D**igital **E**ngineering for **N**uclear **S**afety (**HARDENS**) project. The goal of HARDENS is to provide to the NRC expert technical services in order to (1) develop a better understanding of how Model-Based Systems Engineering (MBSE) methods and tools can support regulatory reviews of adequate design and design assurance, and (2) identify any barriers or gaps associated with MBSE in a regulatory review of Digital Instrumentation and Control Systems for existing Nuclear Power Plants (NPPs).

In the HARDENS project Galois will demonstrate to the Nuclear Regulatory Commission (NRC) cutting-edge capabilities in the model-based design, validation, and verification of safety-critical, mission-critical, high-assurance systems. Our demonstrator includes high-assurance software and hardware, includes open source RISC-V Central Processing Units (CPUs), and lays the groundwork for a high-assurance reusable product for safety critical Digital Instrumentation and Control Systems systems in NPPs.

Before describing our detailed *Technical Plan* in Section 2, in this section we first:

- explain our organization, its background, history, ethos, and areas of expertise (Section 1.2);
- summarize our interest in supporting the the NRC (Section 1.3);
- describe our perspective in high-assurance model-based engineering, particularly from the point of view of specifying and verifying safety, correctness, and security properties and our experience in evaluation and certification regimes that leverage formal assurance comparable to those needed for regulatory reviews (Section 1.4);
- detail our background in high-assurance safety-critical, mission-critical, secure systems engineering—including hardware, firmware, software, and systems engineering using Model-Based Systems Engineering (MBSE)—what we call Rigorous Digital Engineering (RDE) (Section 1.5);
- summarize our perspectives on mainstream model-based engineering and the capability gap between those kinds of systems and technologies and that which is necessary for the NRC (Section 1.6),
- summarize some of our past projects of comparable complexity and import that used model-based engineering (Section 1.7),
- summarize what kinds of tools we have experience with and typically use for these kinds of engagements and what we will be using on this particular project given its constraints (Section 1.8); and
- explain what we hope to do with/for the NRC moving forward beyond this particular project (Section 1.9).

1.2. An Introduction to Galois

Galois is a private, employee-owned research and development organization. Galois was founded in 1999 by academics from an Oregon university, and initially focused on developing tools to support high-assurance cryptographic software for the National Security Agency (NSA).

Our main office is in Portland, OR, and we also have offices in Arlington, VA and Dayton, OH. We have just over 130 employees, and a large fraction of our technical staff have PhDs in computer science. Many of our staff also have graduate degrees in mathematics, electrical engineering, or computer engineering.

While our main clients are Department of Defense (DARPA, Army, Navy, Air Force, Marines, Space Force) and Intelligence Community (NSA, IARPA, etc.) agencies, we also have numerous other government (DHS, DoT, NHTSA, etc.) and commercial (e.g., Amazon and Intel) clients.

Our main areas of expertise include formal methods, formal verification, cryptography, programming and specification languages, theorem proving, software/firmware/hardware correctness and security, semiconductor design, computer security, cyber-physical systems, machine learning and data sciences, mobile device security, and human-computer interaction. Many of these areas are rolled together into the HARDENS project PI's main research area, Rigorous Digital Engineering (RDE).

We operate akin to a computer science and mathematics department, though with a focus on solving our clients' most challenging problems, typically through the invention and creation of new concepts, mathematics, languages, and demonstrable and transitionable tools for others' use. The tools and technologies we create solve unsolved problems and are meant to complement existing solutions and technologies that are available from industry or academic teams.

We release many of the technologies we create as open source, and most of our open source projects are hosted on GitHub. At the time of this writing, our GitHub organization hosts well over 400 projects. These projects are used by hundreds of other organizations for research and development, including those with which we regularly collaborate and compete.

Galois's commercialization and United States Government (USG) transition plan for *High Assurance Rigorous Digital Engineering for Nuclear Safety* pursues two paths as part of our broader commercialization strategy:

- incubation and launch of a new Galois spin-off company (code-named *T00L*) offering consultancy, services, tools, and licensed technology in high-assurance rigorous digital engineering both via products (tools and libraries) and as 3rd-party IP for licensing; and
- direct licensing of Galois's synthesis and formal verification tools to systems, hardware, and software engineering firms (such as Electronic Design Automation (EDA) and software tool vendors, Defense Industrial Base (DIB) primes, and semiconductor companies).

Government transition runs in parallel with our broader commercialization strategy. Galois has a track record of transitioning government-sponsored research and development into practical tools and products for both commercial and government entities. We spin out daughter companies when we witness market demand for a development or support organization around a particular productized set of capabilities. We have launched the following companies over the past nine years:

- **Tozny** A company developing a commercial SDK for end-to-end encryption
- **Formaltech** A company that commercialized the cyber-deception tool CyberChaff and built a reseller channel
- **TangramFlex** A company delivering a workflow engine to re-engineer Department of Defense (DoD) systems into composable, assured, and reconfigurable systems
- **Free & Fair** A company focused on developing trustworthy high-assurance elections technologies to increase faith in democracies
- **MuseDev** A company focused on applying static analysis tools for DevOps to find vulnerabilities in production software (acquired by Sonatype, Q1 2021)
- **Niobium Microsystems** A company focused on the design of high-performance, high-assurance, low-energy semiconductor systems, using novel circuit design flows and leveraging Galois-developed tools and/or IP.

Our first and primary product, originally created exclusively for the NSA, is called *Cryptol*. Cryptol is a model-based executable formal specification language that is used to specify bit-level precise algorithms and specify and prove their properties. It is mainly used to specify and reason about cryptographic algorithms and protocols, but can be used to specify any kind of algorithm. In 2016 Galois released Cryptol version 2 as an open source project and product.

We have developed next-generation systems engineering specification languages (such as **Lando** and **Lobot**), reasoning tools (such as Cryptol, *SAW*, and *Crux*), development platforms (such as the BESSPIN Tool Suite (BTS)), and many other technologies through multiple programs for the DoD and Intelligence Community (IC). These tools are used for a myriad of purposes, including:

- to formally verify that software or firmware programs fulfill their specifications;
- to formally verify that hardware designs fulfill their specifications;
- to formally verify that programs written in different languages (even across the hardware/software boundary) are equivalent;
- to improve the assurance of software using symbolic testing;
- to provide libraries for symbolic formula representation and solver interaction;
- to analyze binaries in a variety of formats and for a host of different Instruction Set Architecture (ISA)s (ISAs);
- to perform binary analysis and rewriting for a variety of purposes;
- to automatically generate model-based tests for a software, firmware, or hardware system;
- to specify systems architectures at a high level leveraging Natural Language Processing (NLP) technology and formal refinement;
- to specify and reason about product lines of hardware, firmware, and/or software systems;
- to extract formal models—including behavioral and architectural models—from source code and binaries;
- to reason about products derived from product lines, particularly automatically generated Central Processing Units (CPUs) and Systems-on-Chip (SoCs); and
- to reason about non-behavioral properties of models or implementations, such as security proofs of cryptographic algorithms and protocols, safety and progress properties of concurrent or distributed systems, information leakage properties of embedded systems and hardware designs.

1.3. Our Interest in Supporting the NRC

Much of Galois's work focuses on *public good R&D*. We do not do R&D for weapons systems, including cyber-offensive weapons, and we will not facilitate the discovery and perpetuation of weaknesses in technology that can be used by bad actors, foreign or domestic, government or criminal.

Consequently, we have a great deal of interest, both as an organization and as individual contributors, in ensuring that the NRC receives the very best information and advice about technologies relevant to NPPs. In the main, we are very concerned about the environment and the future of energy for mankind and believe that nuclear power is a necessary component of that future as we become less dependent upon traditional polluting and finite resources like gas, coal, and oil.

NPP systems of today and the future are, in some sense, in our technical and motivational sweet spot. They are some of the most safety-critical systems in existence today. If we are going to see more NPPs, and if their technological underpinning are modernized by permitting the use of more modern hardware and software (perhaps using MBSE), then we have a moral obligation to help ensure that these systems are safe, correct, and secure. Since Galois's expertise sits at the intersection of those three concerns, and since neglecting any one of those dimensions is unacceptable, we gladly offer our services to help the NRC ensure that future digital systems in NPPs are safe and secure.

1.4. Properties, Engineering, and Evidence

The three kinds of properties necessary for a safety/mission-critical secure system mentioned above—*safety*, *correctness*, and *security*—are coupled to the four core facets of our systems building and evaluation expertise—*hardware*, *firmware*, *software*, and *systems engineering*—and are concretized in how Galois performs Rigorous Digital Engineering (RDE). To dig into the nature and capabilities of RDE, we will first discuss the aforementioned *properties* and the means by which we specify, validate, and verify them using models. Second, we will discuss the different kinds of *rigorous model-based engineering* we perform, advise and teach about, and enrich through invention.

1.4.1. Safety, Correctness, and Security Properties

When we specify a system, our specification typically includes many different semi-formal and formal models. These models often include a domain engineering model, requirements, a system and product architecture, a feature model for the product line, a model-based design of the hardware and software, and an extensive set of properties of the system, its subsystems, and their components.

Domain engineering models describe the concepts of the domain and their inter-relationships at a high level. Ensuring that a domain engineering model is complete and consistent helps ensure (1) that all semi-formal requirements written in English can refine to more formal models, and (2) requirements are consistent and complete. We typically write domain engineering models in **Lando** and in Higher-Order Logic (HOL) in a logical framework like Coq, Lean, or PVS.

Requirements are written in semi-formal English and in a formal notation leveraging the underlying domain engineering model. Thus requirements are often also written in a first-order subset of HOL in the chosen logical framework. We also have experience with using dedicated tools like DOORS, FRET, and others to write and reason about requirements.

Product line architectures are written as feature models describing all of the possible (hardware, software, behavioral, etc.) features of the system and their inter-relationships. We either use the aforementioned HOL model to describe such, or dedicated specification languages for feature modeling, such as Clafer and **Lobot**.

System architectures are described semi-formally and formally using one or more system specification languages. Our commonly used languages include Architecture Analysis and Design Language (AADL), **Lando**, System Modeling Language (SysML), and, again, and enriched model in HOL.

The model-based description of a software system is typically described in several specification languages concurrently. The choice is language is usually dictated by the programming language and platform chosen for the system. Typical languages used include the ISO ANSI C Specification Language (ACSL), Code Contracts, Cryptol, Java Modeling Language (JML), and the SPARK contract language.

Model-based descriptions of hardware designs are also described with Hardware Description Language (HDL)-specific assertion languages. Our main languages that we use include Bluespec SystemVerilog (BSV), Cryptol, SystemVerilog Assertions (SVA), and Universal Verification Methodology (UVM).

Finally, properties are either behavioral properties—denotational specifications of the behavior of the system (e.g., initially, pre- and post-conditions, and invariants)—or non-behavioral properties—cyber-security properties, risks, power, performance, and resource objective functions, reliability, etc. The aforementioned source-level specification languages—such as Cryptol, ACSL, and JML—often give us enough expressive power to specify all but a few safety and correctness properties. Security properties are specified in **Lando**.

The specification we propose to create for the HARDENS project includes the use of a strategic subset of the above technologies. Our choice of technologies is driven by the eventual need for rigor to ensure that the assurance case of the system, particularly for safety properties, is unassailable.

1.4.2. Rigorous Model-Based Engineering

But what does "rigor" mean in this context? In the world of commercial model-based engineering, one group's rigor is another's informality.

At Galois, "rigor" means that **a specification has a precise, unambiguous, formal semantics grounded in real world formal foundations and systems engineering artifacts, such as source code and hardware designs**. Rigor means that models connect to code, to speak, and have real meaning. Two people writing or reading a rigorous specification have zero disagreements about its meaning, because its meaning is grounded in the language of mathematics or logic, not informal English specifications or convention.

Based upon our decades of experience, we argue that only by leveraging rigor—divorced from human fallibility and subjectivity—can we design and build high-assurance safety- and mission-critical systems, such as those needed by the NRC.

Moreover, such rigor is not only of use to the team building and assuring the system, it is enormously valuable to those vetting, analyzing, reviewing, certifying, validating, or inspecting the system. For example, for the national security-related systems that we help create, the existence and independent verifiability of formal specifications, proofs, and other forms of rigorous evidence are at the root of high-trust certification within the DoD and IC. We suggest that this very same kind of evidence and independent verifiability can and should be at the root of NRC review process for future NPP Digital Instrumentation and Control Systems systems.

1.4.3. Rolling Properties and Engineering Together

In order to ensure that the various models created to fully specify a system's structure, behavior, security, etc., there must be a means by which to compose all of the models and ensure that they "fit" together and are consistent.

The Secret Ninja Formal Methods (SNFM) development process and methodology that we have created and refined over the past twenty-five years provides and process- and method-centric mechanism reinforcing this goal, but does not guarantee that all specification artifacts fit together. Intelligent use of well-tuned collaborative development environments (such as our proposed environment for the HARDENS project, GitLab), integrated development and verification environments (e.g., Eclipse or Visual Studio Code), and continuous integration environments (such as our Hydra and Jenkins servers and GitHub Actions) is also a piece of the puzzle.

But we find, at its core, the main mechanism for ensuring that all specifications, development artifacts, and assurance cases fit together is to empower all team members to automatically check, at any point in time as they work, all properties specified about the system, implicit and explicit. This means that, at a twitch of the hand or a click of the mouse, it must be possible to measure and vet every specification, every relevant piece of code, and every relevant assurance artifact against a set of project-, team-, and client-specific properties and objective functions.

In our integrated environments many such checks happen as we type (e.g., static analysis tools that run asynchronously in Eclipse), others happen when we ask for the system to quickly vet our work (e.g., a "check" target in a makefile that takes under 10 seconds to complete), others take place when we have a spare moment to sit back and reflect upon our work thus far (e.g., a "verify" target is a build script that takes 5–30 minutes to execute), others happen as pull/merge-request checks in our Continuous Integration (CI)/Continuous Verification (CV) server (these can sometimes take hours for projects that have extensive assurance cases), and finally the most expensive and comprehensive checks are run automatically overnight on compute servers.

So rolling properties together means that a team is using, in the main, the same process, methodology, core set of tools (i.e., editors and Integrated Development Environments (IDEs) might vary) with continuous engagement with the assurance artifacts of this verification-centric Rigorous Digital Engineering (RDE) development style [50].

While this sounds heavyweight, we find that time and time again products built this way have not only extraordinary assurance, but require comparable amounts of engineering effort and cost as compared to traditional engineering approaches with little-to-no assurance.

1.5. Rigorous Digital Engineering at Galois

Digital Engineering at Galois is grounded in the use of *applied formal methods* to develop hardware, firmware, and software systems. By *applied formal methods* we mean a practical methodology, focused on "real world" systems, whose techniques and tools are explained in mathematics [19]. The techniques of a formal method help construct and analyze specifications and transform/refine specifications into computational artifacts, be they hardware, firmware, or software. By *rigorous systems engineering* we mean the formal, rigorous, and/or systematic application of applied formal methods to reason about a system under development that includes hardware, firmware, and software components [48]. Galois's capabilities enable us to guarantee the trustworthiness of systems where failure is unacceptable.

Our approach and results are unique because we use practical and pragmatic applied formal methods to drive co-design, co-implementation, and co-verification of hardware, firmware, and software implementations (hereafter simply referred to as *implementations*) in a coherent, overarching, integrated fashion, integrating the rigorous engineering methods of Kiniry (the PI) and Zimmerman (a research scientist at Galois) with those of the HATS project at Chalmers [15, 50]. Additionally, rather than specifying, implementing, and reasoning about single products, we specify, reason about, and implement entire *product line families*, helping our clients find the optimal products for their requirements [29]. Finally, as explained below, we are able to formally reason about models of systems (such as the RISC-V Instruction Set Architecture (ISA) itself, accelerator designs, firmware and low-level software behavior, and fault-tolerance and security properties), firmware and software implementations at the source and binary level, and the relationships between these models and implementations [35].

1.5.1. Product Line Engineering

A *product line* is an integrated set of hardware, firmware, and software systems engineering artifacts—including documentation, specifications, source code, and more—that can derive multiple different products through a build/synthesis/fabrication process [4]. *Product line engineering* focuses on identifying, reasoning about, and manipulating the system architectures and features that underlie a product line to introduce variability across the product line family [46]. For example, each specific cryptographic block cipher accelerator that we create is a product with many *features*: it has a specific, concrete algorithm, key width, block size, and mode.

Furthermore, the non-behavioral requirements—including the product's threat model—vary considerably across products, and are also represented in the product line as requirements. Must the cryptographic module be low power? High-performance? How much throughput and latency? Must it be free of side channels? Which kinds of side channels matter? Timing? Electromagnetic emissions? To what degree?

That single product is derived from a *product line family*; the hardware, firmware, and software artifacts used to generate that particular cryptographic module and, typically, its associated firmware, can also generate many other cryptographic modules with different features, both those inherent in the design (such as cryptographic algorithm choices) and those derived from the build processes (such as clock speed and silicon area use) [35].

1.5.2. Galois Model-Based Engineering Capabilities

Our tools and development framework facilitate: (i) formal model-based specification, testing, and reasoning; (ii) compilation and synthesis of implementations and their associated assurance artifacts; (iii) traceability from high-level domain concepts and requirements written in English to transistors in an Application-Specific Integrated Circuit (ASIC); (iv) use and reuse of existing commercial off-the-shelf (COTS) hardware, firmware, and software development tools, especially for assurance; (v) compositional development of and reasoning about subsystems and components, including ASIC domain-specific architectures (for hardware) and modular design of and reasoning about safety-/mission-critical source and object code (for firmware and software); (vi) low-energy circuits capable of operating at arbitrary voltages (from process threshold to standard high-power) and in extreme environments such as those seen in space-based applications, namely with high-energy SEEs and poor thermal conditions; (vii) the ability to lift formal models from existing implementation and assurance artifacts in order to reason about the refinement relationships between models and "real" implementations, regardless of their origin; (viii) the development of high-assurance variants of Commercial Off The Shelf (COTS) compilers, linkers, runtimes, device drivers, kernels, hypervisors, and more to meet unique client requirements; and (ix) the design and creation of new domain-specific languages (DSLs) and domain-specific architectures (DSAs), complete with all artifacts required for others to use our tools, process, and methodology to understand, maintain, extend, and evolve rigorously engineered systems.

Here, we characterize some of these capabilities and explain how we combine them in our RDE process and methodology. In Section 2, we describe exactly what subset of these technologies we intend to apply for the NRC in the HARDENS project.

I. History in High-Performance, High-Assurance Cryptography Galois was started just over twenty years ago by academics interested in the question of how to precisely, formally specify and reason about cryptographic algorithms. Our very first product, Cryptol, was developed with funding from the National Security Agency (NSA) starting in 1999. Early in Galois's existence we focused on research in high-assurance cryptography using functional programming and advanced programming languages like Haskell. In the intervening years, we have grown far beyond this original framing.

Obviously, a lot of cybersecurity-related R&D depends upon cryptographic underpinnings. Therefore, we have developed several tools (such as Cryptol and Software Analysis Workbench (SAW)) for the NSA that help specify, reason about, and generate cryptographic algorithms and their software and hardware implementations. Our first hardware cryptography project, funded by the NSA, focused on compiling Cryptol specifications into optimized, high-performance, high-assurance Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) implementations for Field Programmable Gate Arrays (FPGAs) in circa 2007/8.

However, in order to ensure that real-world systems are trustworthy, it is necessary to focus on more than just software, more than just functional programming languages that only a small fraction of the world uses or understands, and more than just cryptography. Security is a pervasive, non-compositional, system-wide notion, so much of our R&D over the past decade has involved operating systems, hypervisors, networking, cyber-physical systems, embedded systems, and even hardware design and verification.

II. Formal Model-based Specification, Testing, and Reasoning In order to reason about this larger world of systems, networks, hardware, and software, we use a variety of formal methods, formal specification languages, and reasoning tools to work with models. We have expertise in many formal methods such as Z, Vienna Development Method (VDM), B and Event-B, and RAISE. We also use a number of modern methods and languages, such as Coq, Isabelle, PVS, Alloy, F*, Ivy, and our very own Cryptol and SAWscript (and many others). In general, we try to choose the right set of methods, languages, and tools for each major project based upon the client's requirements and goals, as we do in our public projects for major corporations, such as our work verifying Amazon Web Services (AWS)'s cryptography to the benefit of billions [14, 16]; we have experts in nearly every method, language, and tool in use today. Our use and development of formal methods is, in part, what drives our verification tools R&D for the NSA, as discussed below.

III. Compilation and Synthesis of Implementations and their associated Assurance Artifacts While our development style for hand-written code is refinement-centric and leverages concepts from Design by Contact [31], we recognized many years ago the challenges of hand-writing secure, correct code. Consequently, we automatically generate many of our most critical subsystems and components using COTS and custom synthesis and compilation tools that transform mathematical formalizations (in Coq, Cryptol, and SAW) into high-performance, correct-by-construction, formally verifiable implementations.

Complementing these implementations, we automatically generate implementation test benches that leverage popular COTS unit, subsystem, integration, and QuickCheck-style testing frameworks for software and firmware, and custom, UVM-based, or formal verification-centric (e.g., JasperGold and OneSpin) verification benches for hardware. These generated assurance artifacts are integrated into our continuous integration/deployment/verification (CI/Continuous Deployment (CD)/CV) system for continuous assurance, reporting, and risk management.

IV. Traceability from High-Level Domain Concepts and Requirements written in English to Transistors in an ASIC How can one know whether the system that one is specifying, writing, generating, and verifying is the **right** system? This validation question sits at the root of another set of R&D capabilities that will be applied in NRC systems' regulatory reviews.

Since our rigorous engineering process is refinement-based [3, 43], we have defined refinement relations between several system layers and views [34]. The endpoints of these refinement relations include (a) high-level models expressed in precise English that we can parse with domain-specific type-logical grammars [10], (b) formal higher-order models represented in languages like Coq and PVS, (c) formal first-order models represented in languages like Cryptol, SAWscript, Alloy, and SMTLib, (d) software and firmware implementations written or generated in languages like C, Rust, Java, and Haskell, and (e) hardware implementations written or generated in languages like Chisel, Bluespec SystemVerilog (BSV), and SystemVerilog (SV). The behavioral specifications and properties of those implementations are part of these refinement relationships; e.g., a *requirement* in a domain model is refined to (a set of) *properties* in a higher-order model, which are in turn refined to *properties* in a first-order model, which are again refined to first-order *properties* in the implementation and its runtime verification and formal verification test benches.

V. Use and Reuse of Existing COTS Hardware, Firmware, and Software Development Tools, Especially for Assurance Not all development can be green field. This is true in even the most modest embedded and hardware engineering projects. So, as emphasized above, we choose the right tool for each job—not only specification and assurance techniques, but also development tools. We license and use commercial tools (such as Electronic Design Automation (EDA) and systems engineering tools) and also use robust academic and open source tools (such as model checkers, solvers, and theorem provers) when appropriate. Our rigorous engineering methodology is designed to smoothly incorporate new formal methods and tools, as we have done now for some twenty years across several generations of languages, platforms, and frameworks.

VI. Compositional Development and Reasoning of Subsystems and Components, and Module Design and Reasoning about Safety-/Mission-Critical Source and Object Code Additionally, one nearly never has the resources or remit to build the entirety of a high-assurance system. The vast majority of the time, we must integrate with technologies developed by others. Consequently, nearly all of our tools and our rigorous engineering methodology focus on compositional design and reasoning. This includes both our firmware/software design method and our asynchronous hardware design flow. This underlying modularity permits us to decompose system designs, implementations, and assurance across our team(s), facilitates technical and social scalability within and across teams, and helps us use reasoning tools that typically have scalability challenges (such as explicit state model checkers and SAT/SMT solvers) to greater effect than typical, given that our development artifacts are verification-centric from day one [50]. Also, many of our development tools permit us to reason about both source code and object code (see below) in a modular fashion.

VII. Low-Energy Circuits Capable of Operating at Arbitrary Voltages and in Extreme Environments A circuit's energy efficiency is directly proportional to its supply voltage. Modern ASICs use voltage scaling to take advantage of this by adjusting the supply voltage as workloads vary. Traditional synchronous voltage scaling approaches are often hampered by limited feasible voltage ranges and significant transition times when changing voltages. Our proprietary asynchronous circuit technology enables aggressive voltage scaling down to near threshold supply levels, maintains higher performance compared to existing techniques, and allows continuous operation while making voltage adjustments. This eliminates the critical performance penalties normally incurred when adjusting operating voltages [25]. The timing resiliency enabled by this technology can also harden circuits against harsh environmental conditions, such as extreme temperatures, unstable power supplies, or radiation.

Asynchronous circuits also lend themselves well to the compositional designs and high level languages we seek to build [8, 41]. Self-timed designs have gained some traction in modern SoCs, where synchronous top level timing closure has become an exceedingly complex problem [42]. However, industry use of asynchronous designs has historically been limited by reliance on carefully crafted, custom automatic synthesis and place-and-route tools and full custom circuit designs [6, 7]. Our automated CAD flow relies on industry standard design and sign-off tools, standard cell libraries, and a very limited set of custom-designed gates. Variations of this flow have been used on four tapeouts, the most recent of which were in a 12nm process.

VIII. Lifting Formal Models from Implementations and Assurance Artifacts and Reasoning about Refinement Relationships between Models and Implementations In order to reason about our own and others' "real" COTS code, we must be able to reason about "real" languages and object code. As such, many of our tools consume implementations and assurance artifacts (such as test benches) and lift those artifacts to formal models. For example, SAW can consume Low Level Virtual Machine (LLVM) Intermediate Representation (IR), and thus can reason about many languages that compile to LLVM IR, such as C, C++, and Rust [12]. Other tools we have built can consume Chisel, BSV, and SV, and our binary reasoning tools can consume x86, ARM, and (to some degree) RISC-V object code.

Most commonly, to reason about refinement relationships between models and implementations, our formal methods tools automatically reason about the semantics of C source code. This allows us to prove that program transformations preserve critical properties, such as the program's initial semantics; do not violate other critical properties, such as bounds on worst-case execution time; and stay within formally specified bounds on stack, heap, and register usage [20].

IX. High-assurance Variants of COTS Development Tools and Platforms Many tools we have created for clients in the past are built on top of existing frameworks or compilers. For example, we very commonly use, extend, and reason about various operating systems (e.g., Linux and various RTOSs), compilers (C/C++

compilers like Gnu Compiler Collection (GCC) and LLVM, and compilers for more esoteric languages like Rust, Haskell, and others), and platforms (COTS and custom embedded hardware). Consequently, designing, developing, validating, and verifying this demonstration platform for the NRC, and providing evidence that the resulting system is secure, is a familiar landscape.

X. Domain-Specific Languages and Domain-Specific Architectures Each R&D program presents a new set of challenges. Sometimes off-the-shelf formal methods, programming and specification languages, and platforms are effective and efficient. Other times, it becomes clear that a domain-specific approach is better suited to achieving client goals. On these occasions, we work with the client to design solutions specific to their challenges, often in the form of domain-specific programming and specification languages and domain-specific architectures and accelerators. Our work on formally verified cryptographic extensions and formally assured measured boot for a RISC-V System-on-Chip (SoC) have followed exactly this trajectory [35]. In that project we used Cryptol and **LANDO** specifications of the extension to synthesize formally assured cryptographic Intellectual Property (IP) blocks for the SoC.

1.6. Model-Based Engineering in the Mainstream

As mentioned in the RFP, the term Model-Based Engineering (MBE) is used in the mainstream in a variety of ways by software engineering organizations, tool providers, and consultants. In the main, there are dozens, if not hundreds of technology stacks that provide some means by which to write and use models of a system.

Some models are written in de facto standardized languages like Unified Modeling Language (UML), SysML, and AADL. Some kinds of models, such as a subset of the model types available in UML (e.g., class, package, and sequence diagrams), can be extracted from source code using static or dynamic analysis. Some kinds of these same kinds models can be used to generate source code, which is typically a one-way process and consists only of code skeletons (modules, header files, some type declarations, but no function bodies).

Very, very few model-based code generation frameworks provide any assurance at all about their generated code. Most do not even generate test cases for the implementation, do not use safe programming languages, and do not guarantee that the generated code will even compile correctly. Moreover, in the vast majority of situations, if the source code is modified the model-code correspondence is no longer valid, and cannot be enforced.

Vanishingly few model-based engineering tools provide any means by which to reason **about** models. Usually one can check that a model is well-typed/formed. Sometimes one can measure the complexity of a model by some means. Occasionally one can simulate a model and manually review whether or not its behavior makes sense. It is very rare to see any non-trivial assertions written about models, e.g., in Object Constraint Language (OCL) [45] about UML models or Behavioral Language for Embedded Systems with Software (BLESS) for AADL models [38, 45].

Consequently, the typical model-based software, hardware, and systems engineering tools have little value when it comes to high-assurance engineering. We have found through years of experience that using them to specify and generate (part of) a system and then trying to provide assurance for the resulting implementations is extremely difficult, expensive, and risky.

Commercial design or implementation languages that demonstrate a real dedication to assurance of model-based designs and implementations include Ansys's Safety Critical Application Development Environment (SCADE) tooling [1], Praxis's SPARK toolset [11], and NASA's bespoke tools for model-based systems engineering [13]. High-end, verification-centric hardware design methodologies also fall into this camp [44].

Our R&D focus on high-assurance model-based engineering is grounded in this background. One must understand the capabilities and limitations of today's commercial and research technologies in order to create new languages, tools, and technologies for tomorrow's high-assurance safety- and mission-critical secure systems.

1.7. Past Rigorous Digital Engineering Projects

In Section 5 we summarize some recent examples of our use of Digital Engineering (DE)/MBSE at Galois in projects whose domains overlap with those systems the NRC wants to regulate, insofar as they involve high-assurance ASIC and FPGA design, systems engineering, and software engineering. In particular, we summarize three projects there that are further evidence of our capabilities. All of these projects use the Galois Rigorous Digital Engineering (RDE) methodology, which is, in turn, based upon the PI's work in SNFM [36]. All of these projects use Model-Based Engineering (MBE) of various kinds, all have digital twins that facilitate the simulation and emulation of the systems at multiple fidelities, all use our continuous integration and verification (CI&CV) infrastructure for DevSecOps, and all use Open Systems Architectures (OSAs).

I. GULPHAAC *Galois Ultra-Low-Power High-Assurance Asynchronous Cryptography* (*GULPHAAC*) was a NSA-funded project focused on the question of how to architect, design, and assure a high-assurance, ultra-low power cryptographic hardware module. The GULPHAAC chip was designed and developed using the Galois RDE methodology.

In particular, all of the implementations of the cryptographic algorithms in the chip (multiple versions of the Advanced Encryption Standard (AES), Simon, and Speck block ciphers) were automatically generated to SV from formal models of those algorithms written in Cryptol [18]. These implementations are comparable to the lowest power and highest performance hand-written implementations of the algorithms ever developed. Moreover, the ASIC is asynchronous, using a clockless design that ensures correct and secure operation with any power supply, from transistor threshold (just over 0.6V) to nominal (just shy of 1.4V).

II. 21CC *21st Century Cryptography* (*21CC*) was a DARPA-funded project that followed in GULPHAAC's footsteps. In 21st Century Cryptography (21CC), Galois was tasked with designing, implementing, and assuring a power side-channel free, ultra-low power, asynchronous cryptographic hardware module. The resulting ASIC includes the AES, Secure Hash Algorithm (SHA), and Hash-based Message Authentication Code (HMAC) algorithms as well as a RISC-V CPU.

Once again, Model-Based Engineering was used to generate hardware implementations from formal models. We used a transliteration of those models' theorems into SVA-based properties as a verification bench for the Register Transfer Level (RTL), verifying the generated RTL using Cadence's JasperGold tool [30].

III. SHAVE In Balancing the Evaluation of System Security Properties Against Industrial Needs (BESSPIN) we developed a set of RISC-V CPUs and SoCs that run on Xilinx FPGAs, both on developer boards and in the AWS cloud. Preceding BESSPIN and System Security Integration Through Hardware and Firmware (SSITH), we executed a seedling for SSITH called *Software/Hardware Assurance Verified End-to-End* (*SHAVE*), which also used a "Soft-core" RISC-V processor. In SHAVE we developed a bump-in-wire AES module that was able to be provisioned with a key once, act only as an encryptor or decryptor device, and whose requirements and formal assurance case ensured that it behaved exactly and only as promised: no more, no less.

Included in SHAVE was a 32-bit RISC-V processor written in the BSV HDL, a cryptographic module that could either be implemented as a hardware coprocessor or a firmware library, and a moderately complex state-machine driven software application layer that drove the entire system. The RISC-V core was assured with RISC-V compliance testing. The hardware implementation of the AES core was synthesized automatically from a Cryptol model and was formally assured with a version of SAW, developed as part of the seeding, that can formally verify the core expression language of BSV. The firmware cryptography is either a hand-written and reused COTS AES implementation or an automatically synthesized C or LLVM implementation that is formally verified with SAW and rigorously validated with runtime tests. The system also includes a formally assured measured boot.

The entire system, including the software application layer, was formally specified with the Business Object Notation (BON) system modeling language[1] (for the system domain model, requirements, scenarios, events, and architecture) and the PVS logical framework; its abstract state machines (Abstract State Machine (ASM)) were written in Cryptol and a Domain Specific Language (DSL) used by the Frama-C verification tool suite. The SHAVE product line is specified with a graphical feature model and is realized through a build system, the correctness and security properties are specified in Cryptol, ACSL, and SAWscript, and verification is accomplished through a combination of a half dozen formal methods tools. We strongly adhered to the Galois RDE methodology in this project.

1.8. Tools Used for Rigorous Digital Engineering at Galois

The tools that we propose to use in the HARDENS project represent only a small subset of those used and created at Galois for RDE. Here we mention the other technologies in our toolbox and their utility in model-based engineering of safety-/mission-critical secure systems. This list is not proscriptive or meant to denote a hard edge of capability. When confronted with a new client challenge, it is not uncommon for us to realize that a tool that is not yet in our toolbox to be well-suited for a project, and thus we use internal resources for professional development in order to add that new tool to our capabilities set. Each of these classes of technologies focuses on those that are suited for high-assurance engineering. There are numerous technologies we regularly use (e.g., the Haskell functional programming language) that are not well-suited for the assurance demands of safety-critical systems. Some technologies are listed in multiple classes as they have multiple uses.

- **Abstract and finite state machine specification and verification**: Aoraï, ASM, Cryptol, and SAW.
- **Architecture or design specification**: AADL, BON, **Lando**, **Lobot**, SysML, and UML.
- **Automated solvers**: ABC, Alt-Ergo, Boolector, CVC, MathSAT, OpenSMT, Yices, and Z3.
- **Cyber-security specification and verification**: BESSPIN Tool Suite (BTS), Cryptol, and SAW.
- **Formal methods with tool support**: Alloy, ASM, B, CafeOBJ, CASL, Event-B, JML, Maude, RAISE, OBJ (and its various flavors), Unified Theories of Programming (UTP), VDM (and its various flavors), and Z (and its various flavors).
- **Hardware design languages**: BSV, Chisel, Verilog, VHDL, SystemC, and SV.
- **Hardware verification technologies**: JasperGold, EBMC, Formality, Kami, SAW, OneSpin, Questa Formal Verification Apps, Upscale QED, and Yosys.
- **Logical frameworks**: Coq, Isabelle, Lean, and PVS.
- **Product line engineering specification and verification**: Clafer, **Lando**, and **Lobot**.
- **Programming languages**: a verifiable subset of C, C++, C#, and Java; Rust and SPARK.
- **Protocol specification and verification**: Cryptol, F*, FDR4, Lustre, Tamarin, Verifpal, and UPPAAL.
- **Requirements specification and verification**: FRET, **Lando**, Requirements State Modeling Language, SpeAR, and PVS.
- **Software/firmware program verification technologies**: ACL2, BLAST, Boogie, CBMC, Crux, Cryptol, Dafny, Frama-C, KeY, SAW, SPARK, Verifier for Concurrent C, VeriFast, Verified Software Toolchain, and Why3.
- **Specification languages for source-level specifications of software source code**: ACSL, Code Contracts, Cryptol, and JML.

[1]BON is an ancestor to **Lando**.

- **Tools or formal methods for specifying and reasoning about distributed, concurrent systems**: BLAST, CPAchecker, Communicating Sequential Processes (CSP), FDR4, Ivy, Java Pathfinder, mCRL2, NuSMV, PRISM, SPIN, TLA toolbox, UNITY, and UPPAAL.
- **Tools for specifying and reasoning about cryptographic algorithms and protocols**: Cryptol, CryptoVerif, EasyCrypt, F*, and ProVerif.

As one can see, we have a huge toolbox of assurance-related technologies at our disposal. Moreover, we are happy to help other organizations make well-informed evidence-based choices about what new technologies they may find useful for their particular team, system, and client. Teaching others to perform RDE is, in fact, the best possible way to ensure that more and more of the systems that society depends upon are trustworthy and secure.

1.9. Future Work Supporting the NRC

We are keen to demonstrate model-based engineering tools that are well-suited for safety-critical embedded systems like the system we propose to create for the NRC in the next chapter, Section 2. We are also happy to help experts at the NRC and its clients learn more about the concepts, formal methods, languages, tools, and technologies in this R&D area.

But our deeper, longer-term desire is to ensure that NPP operators of the future make excellent technology choices in designing, building, and assuring Digital Instrumentation and Control Systems systems of the future. We can foresee a future where Galois, or more likely our daughter firm *T00L*, is tasked with the most difficult and important computer science and computer engineering problems at the root of such a safety- and mission-critical cyber-secure system. This is the future work we would most like to see as we start to walk down a path of working with the NRC.

2. Technical Plan

Our technical plan for the HARDENS project focuses on addressing the NRC's goals for understanding what is possible with MBE today. We focus only on the Reactor Trip System (RTS) and its systems architecture, as required in the RFP, and we will ensure that the desired characteristics (based on requirements set in IEEE Std. 603-2018 [27]) are covered in the demonstrator.

In addition, we also discuss here the state-of-the-art at the very edge of capabilities in commercial and transitional research. This means we also offer technical services—via demonstration, expert assistance, guidance, and training to the NRC—well beyond the mainstream of widely used MBE technologies. Some of these "cutting edge" capabilities that are in use in the DoD, IC, and some Fortune 100 companies for safety- and mission-critical systems can be easily demonstrated in the HARDENS project. Thus, we include them as optional, aspirational goals if there are sufficient resources and NRC mandate in this project.

We already discussed in Section 1.6 why we hold concern about the capabilities, claims, and utility of many of the more "popular" MBE technologies and corporations. We leave it to the NRC to discern hype from reality based upon the evidence presented, and we encourage NRC technical leaders to communicate with technical leaders of the projects we cite in Section 5 at, e.g., DARPA, the NSA, and AWS.

2.1. Implementation

First we will summarize our intentions for the implementation of the RTS system requested in the RFP. We do so by describing the system architecture we intend to create and assure for the NRC. As we do not know if reviewers are experts in SysML, we instead explain our system architecture using generic architecture diagrams and detailed discussion.

Note that we realize most of the assurance characteristics required in the RFP via architectural design choices: putting different implementations of redundant components in separate places in the architecture spread across hardware and software. We explain this partitioning below in Section 2.1.3.

2.1.1. System Architecture

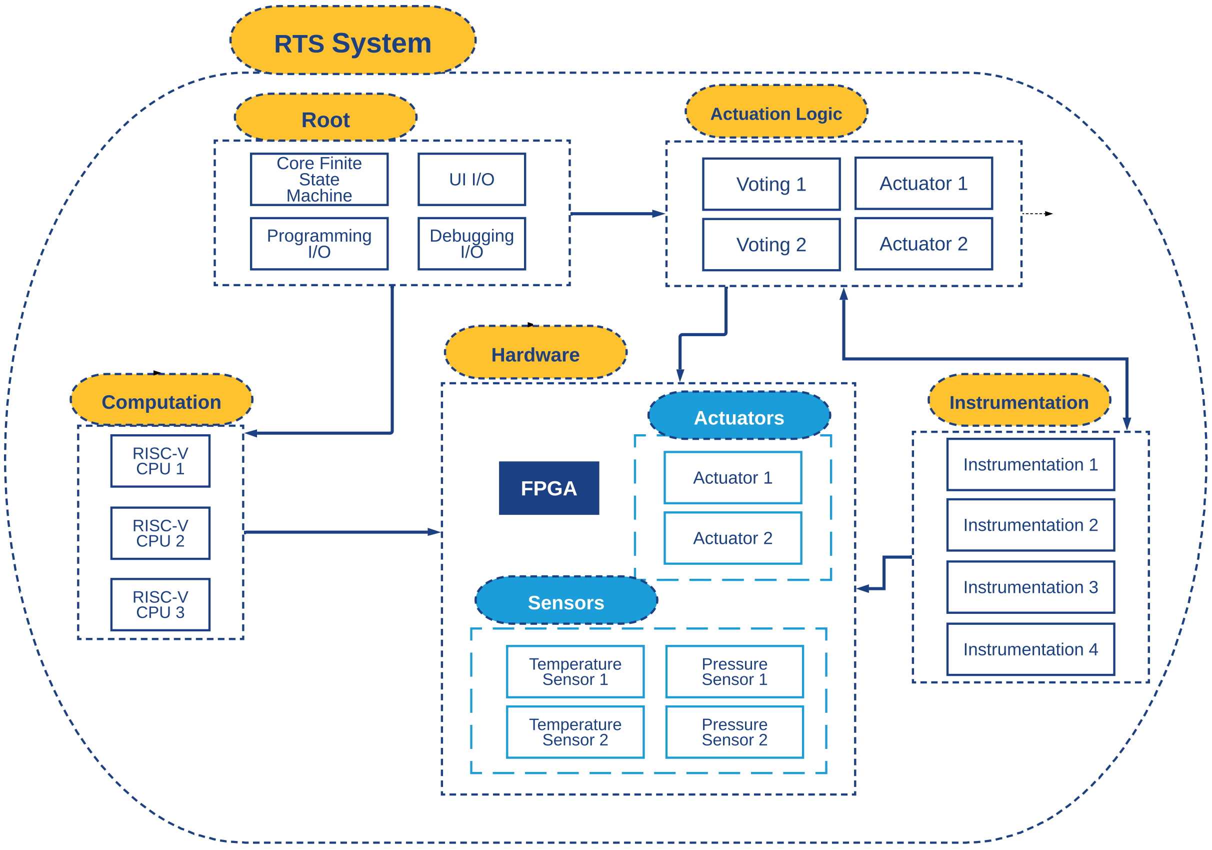

Figure 1: The RTS System Architecture

First, we will review the system architecture of the RTS system. It is depicted in Figure 1, expressed in the graphical **Lando** syntax. Concrete components are expressed in named squashed rectangles with solid outlines. Subsystems are denoted with named squashed rectangles with dashed outlined. All components are in a subsystem, and subsystems can nest. The outer most subsystem represents in this diagram the RTS system itself. Double lines with an arrowhead denote client-supplier relationships; the source of the arrow indicates the client and the arrow end of the line denotes the supplier/dependency of the relationship.

The overall shape of the RTS system is an archetypal *sense-compute-actuate* architecture. Sensors are in the *Sensors* subsystem. They are read by the *Instrumentation* subsystem which contains four separate and independent *Instrumentation* components. The "Compute" part of the architecture is spread across the *Actuation Logic* subsystem—which contains the two *Voting* components which perform the actuation logic itself—and the *Root* subsystem which contains the core computation and I/O components, and the two separate and independent devices that drive actuators.

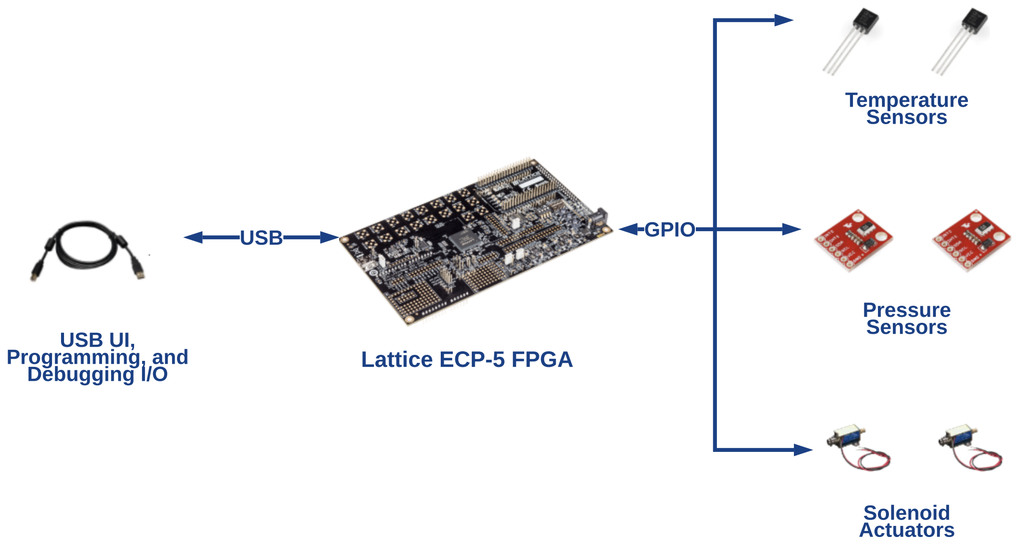

Figure 2: RTS Hardware Artifacts and their Physical Architecture

2.1.2. Hardware Architecture

The FPGA platform that we intend to use is a low-cost (circa $100) COTS Lattice ECP-5 FPGA development board. There are several such board available on the market, one of which is seen here. We choose a Lattice FPGA because (a) they are widely available, (b) they are low cost, (c) they support both commercial and open source FPGA development flows, and (d) they are a favored platform for advanced R&D in open hardware development. There is nothing about our implementations use of the FPGA that is specific to the Lattice FPGA; we could have easily used other FPGA-based platforms with which we are familiar, including those from Xilinx, Intel/Altera, and others.

Attached to the FPGA board via General Purpose I/O (GPIO) are the redundant temperature and pressure sensors of the *Sensors* subsystem and the solenoids of the *Actuators* subsystem. The host development platform used to program, debug, and demonstrate the RTS system is attached to the FPGA via Universal Serial Bus (USB).

2.1.3. Software and Programmable Hardware Architecture

Figure 3 shows the software and Soft-core hardware installed on the FPGA. This is a dataflow diagram. Rounded rectangles are concrete implementation artifacts, either source code or binaries. Squashed rectangles are tools used to transform inputs to outputs. E.g., the CompCert compiler is a formally verified C compiler that we will use to compile C source code to RISC-V object code [40].

We intend to use a Soft-core open source RISC-V 32 bit microcontroller as the CPU platform of the demonstrator (lower left corner of the figure). We choose a RISC-V CPU because it is an open source ISA, several high-quality open source RISC-V CPUs exist, and we know the platform well. Also, because the RISC-V ISA is small, elegant, well-designed, some of these open source CPUs are formally verified, and are thus high-assurance.

Because a RISC-V microcontroller is so small, we are also able to put several of them on our target FPGA. We intend to use three CPUs: one for running the core event processing and User Interface (UI) for the RTS demonstrator, and the other two for independently running two different versions of software implementations

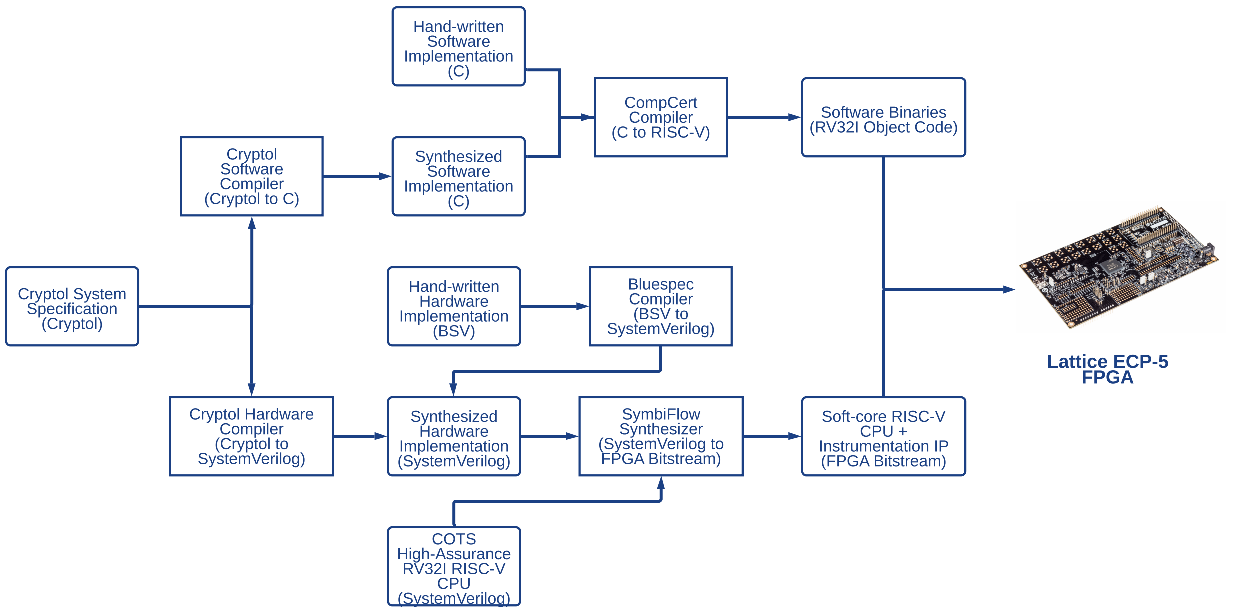
Hand-written Software Implementation (C)
Cryptol Software Compiler (Cryptol to C)
Synthesized Software Implementation (C)
CompCert Compiler (C to RISC-V)
Software Binaries (RV32I Object Code)
Cryptol System Specification (Cryptol)
Hand-written Hardware Implementation (BSV)
Bluespec Compiler (BSV to SystemVerilog)
Lattice ECP-5 FPGA
Cryptol Hardware Compiler (Cryptol to SystemVerilog)
Synthesized Hardware Implementation (SystemVerilog)
SymbiFlow Synthesizer (SystemVerilog to FPGA Bitstream)
Soft-core RISC-V CPU + Instrumentation IP (FPGA Bitstream)
COTS High-Assurance RV32I RISC-V CPU (SystemVerilog)

Figure 3: Dataflow of RTS Implementation Artifacts

of the instrumentation subsystem (upper left of figure). In addition to the RISC-V CPUs on the FPGA, we also will be including two independent hardware implementations of the instrumentation subsystem (bottom left of the figure). We discuss this architecture and our motivations for its design more below.

We write the hardware components of the instrumentation subsystem in two different HDLs. One of these hardware components will be implemented in the BSV HDL. BSV is compiled to SV using the Bluespec Compiler (`bsc`). The other hardware component will be synthesized from Cryptol to SV. Both independent SV components will be composed with the RISC-V CPUs into a single SoC which will synthesized to an FPGA bitstream using the open source SymbiFlow tool suite.

In order to boot the RTS demonstrator, the SymbiFlow tool suite is used to program the FPGA with the bitstream, then the software implementation is loaded into the memory of the RISC-V cores, and the system is started by telling all three cores to start executing their loaded programs.

Note that, because the simplicity of the RTS system, we do not intend to use an operating system at all in this high-assurance demonstrator. Instead, all software components are "bare metal" code that run directly on the hardware.

2.1.4. RTS Instrumentation Systems Architecture

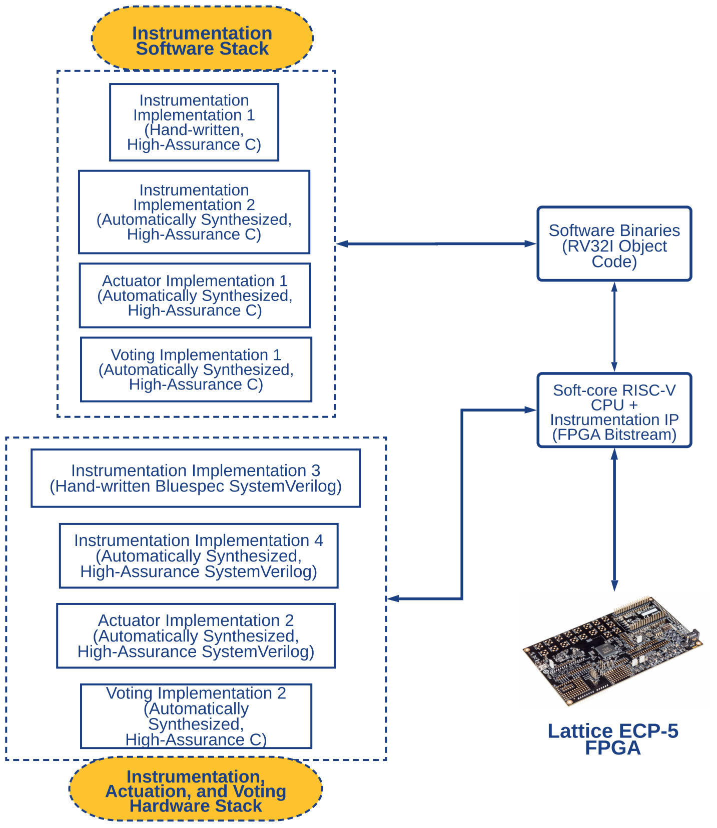
Instrumentation Software Stack
Instrumentation Implementation 1 (Hand-written, High-Assurance C)
Instrumentation Implementation 2 (Automatically Synthesized, High-Assurance C)
Actuator Implementation 1 (Automatically Synthesized, High-Assurance C)
Voting Implementation 1 (Automatically Synthesized, High-Assurance C)
Instrumentation Implementation 3 (Hand-written Bluespec SystemVerilog)
Instrumentation Implementation 4 (Automatically Synthesized, High-Assurance SystemVerilog)
Actuator Implementation 2 (Automatically Synthesized, High-Assurance SystemVerilog)
Voting Implementation 2 (Automatically Synthesized, High-Assurance C)
Instrumentation, Actuation, and Voting Hardware Stack
Software Binaries (RV32I Object Code)
Soft-core RISC-V CPU + Instrumentation IP (FPGA Bitstream)
Lattice ECP-5 FPGA

Figure 4: RTS Instrumentation Architecture

As seen in Figure 4, the RTS instrumentation and actuation subsystems are bifurcated across hardware and software subsystems, each of which has their own independent I/O channel (a GPIO channel) to the platform's sensors and actuators. As each software and hardware instrumentation component has a separate I/O channel for gathering sensor input and driving actuator output, this separates concerns, I/O control, and ensures independent I/O bandwidth and latency for all channels. Note that, since we are demonstrating on a single FPGA, we recognize that the FPGA development board itself is a single point of failure. But since this is just a demonstration system, we believe the complexity of a decoupled, distributed FPGA architecture consisting of five FPGA boards coupled via an inter-FPGA coordination protocol over GPIO is unnecessarily complex. We certainly can build such an architecture, if deemed necessary by the NRC.

Size and Complexity of Architecture and its Implementation

As discussed, RTS architecture has a total of seven I/O components: four components in the instrumentation subsystem, and two components in the actuation subsystem, one UI component on the main CPU. The system has three other components: the ASM that implements the core algorithm of the sense-compute-actuate subsystem, and the two voting components. Additionally, to support demonstration, the RTS system has two I/O reused components—one for programming the FPGA and one for debugging the system—and one reused compute component reused thrice (the RISC-V CPU). Thus, there are a total of thirteen components in the architecture, but three of them are reused and thus do not represent new specifications or code that must be created or verified to demonstrate the RTS system.

Component	Est. Complexity	Est. Spec. Size (LOC)	Est. Impl. Size (LOC)
Core ASM	Medium	100	100
UI I/O	Low	100	100
Instrumentation 1	Low	50	100
Instrumentation 2	Low	50	100
Instrumentation 3	Low	50	100
Instrumentation 4	Low	50	100
Actuator 1	Low	50	100
Actuator 2	Low	50	100
Voting 1	Medium	100	100
Voting 2	Medium	100	100
RISC-V CPUs	Complex	0	reused
Programming I/O	Medium	0	reused
Debugging I/O	Medium	0	reused
Total	**Medium**	**700**	**1000**

Figure 5: Estimated Complexity and Size of RTS Specification and Implementation

Figure 5 summarizes these thirteen components. As one can see, we estimate a total of 700 lines of specification and 1,000 lines of code must be created for the implementation of this demonstrator.

DevSecOps for RTS

We will host the project on Galois's self-hosted GitLab server [24]. We intend to use GitLab's issue tracker to track ideas, features, and bugs. Issues will be labeled in accordance with the system architecture and our normal RDE process. We will use our standard Galois GitFlow-based [23] git repository workflow for sharing artifacts across the team. We will use GitLab CI/CD/CV for continuous integration, and component/unit, subsystem, and system testing, and component formal verification.

We typically offer clients full visibility into our collaborative development environments. NRC personnel will be able to directly see all development and V&V work live, as it happens.

Additional Specifications

Figure 6 summarizes all of the specification artifacts associated with the RTS demonstrator. We describe them from left to right in what follows. Each level of specification is a refinement of the previous, adding additional, more detailed, information about the system. The most abstract specification is written in precise English, and the most precise specifications are written as test benches in all three of our implementation languages: C, BSV, and SV.

Those specifications in with solid (blue) fill are part of our Statement of Work (SoW); those with a solid (green) fill are not, but if we have the resources available in the project after all mandatory requirements are complete we intend to create them and demonstrate their use for assurance to the NRC. We discuss these optional/additional specifications below in Section 2.2.2.

The top-level overarching project specification will be written in precise English in Markdown format and facilitate traceability across the project using hyperlinks. Examples of such top-level detailed "README" system specification are seen in any of our GitHub organization projects, such as the SSITH Government Furnished Equipment (GFE) repository.[2]

[2]See https://gitlab-ext.galois.com/ssith/gfe

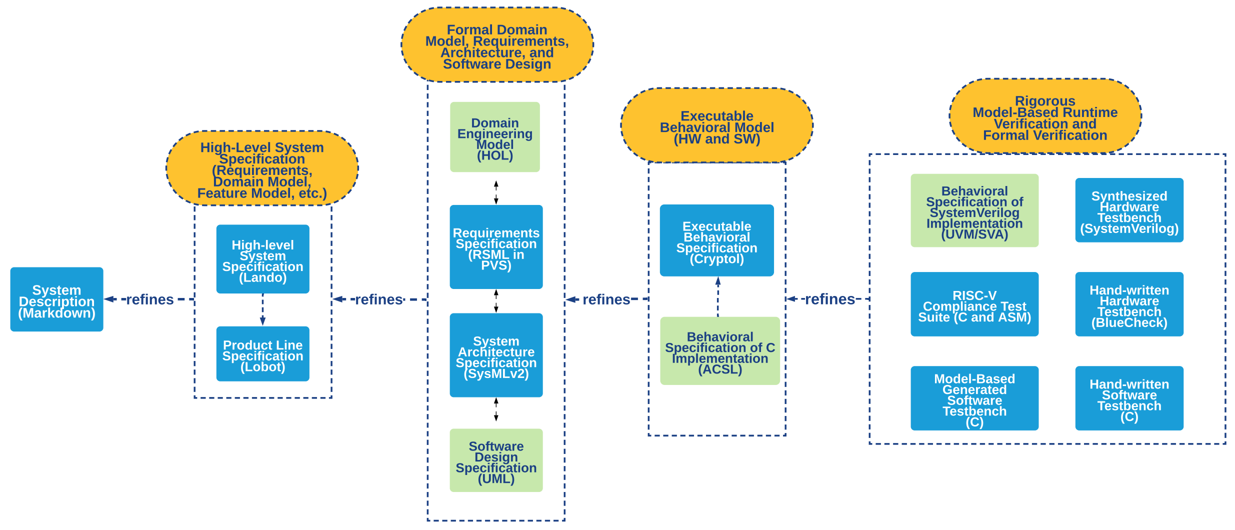
Formal Domain Model, Requirements, Architecture, and Software Design
Domain Engineering Model (HOL)
Requirements Specification (RSML in PVS)
System Architecture Specification (SysMLv2)
Software Design Specification (UML)
High-Level System Specification (Requirements, Domain Model, Feature Model, etc.)
High-level System Specification (Lando)
Product Line Specification (Lobot)
System Description (Markdown)
refines
Executable Behavioral Model (HW and SW)
Executable Behavioral Specification (Cryptol)
Behavioral Specification of C Implementation (ACSL)
Rigorous Model-Based Runtime Verification and Formal Verification
Behavioral Specification of SystemVerilog Implementation (UVM/SVA)
Synthesized Hardware Testbench (SystemVerilog)
RISC-V Compliance Test Suite (C and ASM)
Hand-written Hardware Testbench (BlueCheck)
Model-Based Generated Software Testbench (C)
Hand-written Software Testbench (C)

Figure 6: RTS Specification Artifacts

We intend to use **Lando** and **Lobot** to specify the RTS's high-level system specification and architecture and feature model. The partitioning of functionality across the hardware/software interface, as discussed above, will be specified as a product line configuration in **Lobot**. Example **Lando** specifications can be found, e.g., in the SSITH Voting System repository.[3]

The requirements for the RTS system will be specified in one of several formal requirements languages, such as Requirements State Modeling Language [26], SpeAR [21], or FRET [22]. The choice of language and tool support depends upon feedback from the NRC with regards to comfort levels in formalism. All of these requirements elicitation languages and tools facilitate checking the consistency and completeness of state-based requirements.

The system's architecture will be specified with SysML version 2. We will use the SysML version 2 prototype to specify the architecture in the SysML v2 textual syntax and we will render from such the graphical views of the architecture.[4]

The core model-based specification will be written in Cryptol. The Cryptol model is an executable specification, and will include properties about the system that must hold for the model to be safe and correct. We will use the Cryptol tool to verify that the specification fulfils these properties. We will also use the Cryptol tool's software and hardware compiler back-ends to generate some of the implementations of the RTE system, as discussed next.

The four redundant divisions of instrumentation will be (1) hand-written in C with the intention of conforming to the Cryptol specification, (2) generated in C from the Cryptol specification, (3) generated in SV from the Cryptol specification, and (4) hand-written in BSV. Likewise, the pair of actuation components and pair of voting components will be generated in C and SV from a Cryptol specification.

As mentioned earlier, SV components will be synthesized to a bitstream for the FPGA by using SymbiFlow. Likewise, we will use the CompCert compiler to compile the hand-written and generated C code to RISC-V binaries.

[3]https://gitlab-ext.galois.com/ssith/SSITH-FETT-Voting/-/tree/develop/design
[4]See https://github.com/Systems-Modeling/SysML-v2-Release

2.2. Validation and Verification

Validation and Verification of the RTS demonstrators has two facets: the mandatory V&V demanded by the RFP, and the possible V&V afforded by modern model-based digital engineering. We discuss both of these classes of V&V below.

2.2.1. IEEE Standard 603-2018 Characteristics to be Demonstrated

The characteristics to be demonstrated as itemized in the RFP will be validated and verified by the following means. We label each characteristic with a short name to simplify cross-referencing. We *emphasize* the text of each characteristic that comes directly from the RFP.

- *completeness and consistency of requirements* will be demonstrated by:
  - **[Consistency]** both formal and rigorous consistency checks of the requirements will be accomplished by using false theorem checks and proofs in the Cryptol model and in software and hardware source code; and
  - **[Completeness]** a rigorous completeness validation of the requirements will be accomplished by demonstrating traceability from the project specification (including the RFP text describing the reactor trip system) to the formal models of the system and its properties; and
  - **[Completeness]** a formal verification of completeness of the requirements will be accomplished by using the chosen requirements checking tool, as discussed above;
- **[Instrumentation Independence]** *independence among the four divisions of instrumentation (inability for the behavior of one division to interfere or adversely affect the performance of another)* will be demonstrated architecturally via the decoupling of computation across the two RISC-V instrumentation cores and two instrumentation units running on the FPGA;
- **[Channel Independence]** *independence among the two instrumentation channels within a division (inability for the behavior of one channel to interfere or adversely affect the performance of another)* will be demonstrated architecturally by decoupling the compute and I/O channels of the units from one another;
- **[Actuation Independence]** *independence among the two trains of actuation logic (inability for the behavior of one train to interfere or adversely affect the performance of another)* will be demonstrated architecturally by partitioning the actuation logic across software and hardware units;
- **[Actuation Correctness]** *completion of actuation whenever coincidence logic is satisfied or manual actuation is initiated* will be demonstrated by rigorous validation via runtime verification and formal verification of the model and its implementation, as discussed in detail below; and
- **[Self-Test/Trip Independence]** *independence between periodic self-test functions and trip functions (inability for the behavior of the self-testing to interfere or adversely affect the trip functions)* will be demonstrated architecturally by partitioning the actuation logic across software and hardware units.

2.2.2. Validation and Verification of the Implementation

The set of artifacts that could be created and used in a full, high-assurance, rigorous digital engineering demonstration exercise are considerably larger than those that we promise to create in the Statement of Work. We discuss a typical set of specifications and their use below.

It is not only possible, but we suggest it is mandatory, that any deployed NPP system include only formally assured software, firmware, and hardware. Such formal assurance must not only include safety properties, but full functional correctness and security of all models and their implementations. We discuss below what this full-blown high-assurance case study/product might look like, using all of the appropriate rigorous (model-based) digital engineering tools that are a part of our toolbox.

The only rigorous validation and verification of the implementation demanded by the RFP is the penultimate property specified above: **[Actuation Correctness]**. Thus we intend to use only the *emphasized* techniques and artifacts discussed below in order to validate and verify these properties.

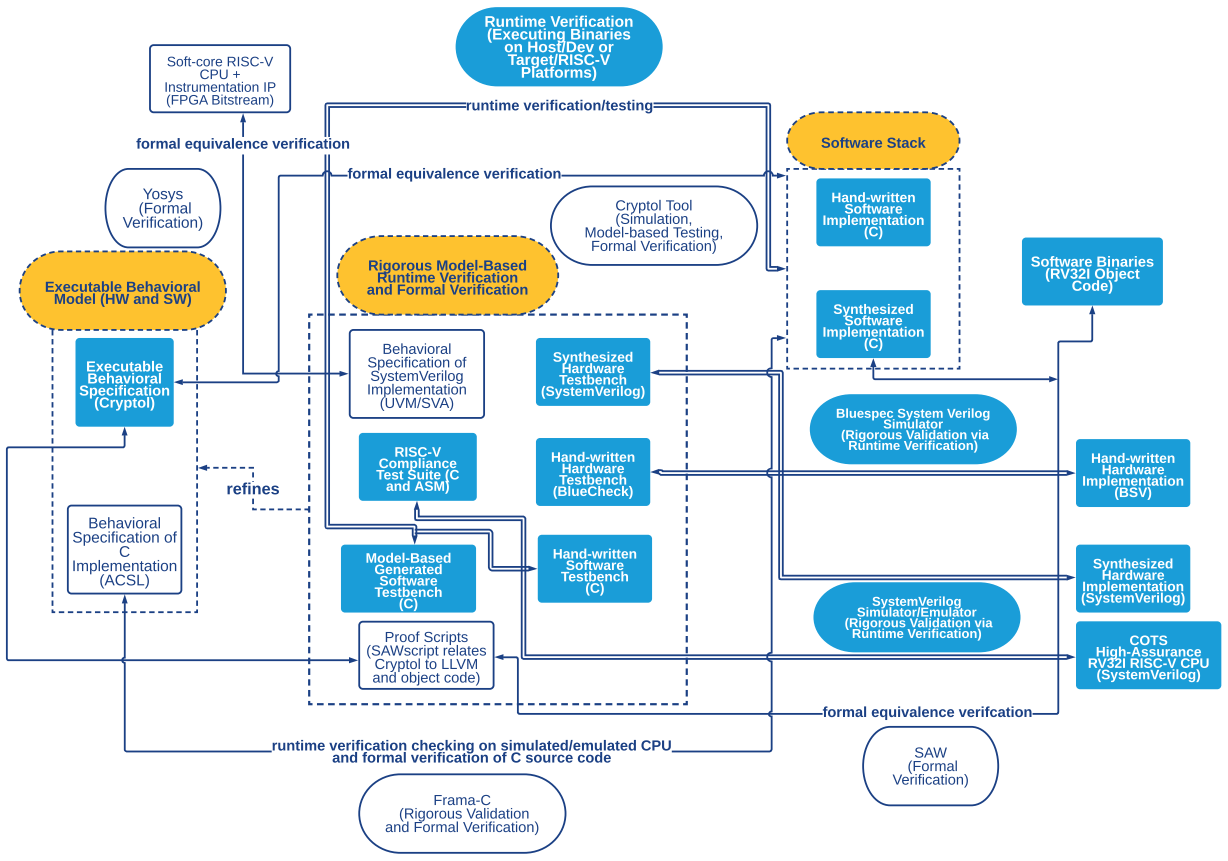
Runtime Verification (Executing Binaries on Host/Dev or Target/RISC-V Platforms)
Soft-core RISC-V CPU + Instrumentation IP (FPGA Bitstream)
runtime verification/testing
Software Stack
formal equivalence verification
formal equivalence verification
Yosys (Formal Verification)
Hand-written Software Implementation (C)
Cryptol Tool (Simulation, Model-based Testing, Formal Verification)
Software Binaries (RV32I Object Code)
Executable Behavioral Model (HW and SW)
Rigorous Model-Based Runtime Verification and Formal Verification
Synthesized Software Implementation (C)
Executable Behavioral Specification (Cryptol)
Behavioral Specification of SystemVerilog Implementation (UVM/SVA)
Synthesized Hardware Testbench (SystemVerilog)
Bluespec System Verilog Simulator (Rigorous Validation via Runtime Verification)
RISC-V Compliance Test Suite (C and ASM)
Hand-written Hardware Testbench (BlueCheck)
Hand-written Hardware Implementation (BSV)
refines
Behavioral Specification of C Implementation (ACSL)
Model-Based Generated Software Testbench (C)
Hand-written Software Testbench (C)
Synthesized Hardware Implementation (SystemVerilog)
SystemVerilog Simulator/Emulator (Rigorous Validation via Runtime Verification)
Proof Scripts (SAWscript relates Cryptol to LLVM and object code)
COTS High-Assurance RV32I RISC-V CPU (SystemVerilog)
formal equivalence verifcation
runtime verification checking on simulated/emulated CPU and formal verification of C source code
SAW (Formal Verification)
Frama-C (Rigorous Validation and Formal Verification)

Figure 7: RTS Validation/Testing and Verification Artifacts

Figure 7 diagrammatically summarizes the formal specifications and their tool-based relationships with implementation artifacts. As before, rounded rectangles represent specifications and implementation artifacts. Squashed rectangles represent computational tools that are used to perform validation and verification. Note that we do not mention our V&V of requirements specifications, as they are one abstraction level "up" from these model-based specifications. Refinement between those specifications and their realization in model-based specifications is accommodated by our refinement-centric methodology, as discussed in several papers [32–34].

Only those artifacts that are filled (and colored in blue) are promised as a part of the SoW. The tools that we intend to use to provide some degree of assurance about our implementations are also filled (and colored in blue). Thus, we intend to automatically generate model-based test benches for our software and SystemVerilog implementations and execute those test benches via runtime verification on both host-based emulation and software and FPGA-based simulation/emulation.

By host-based emulation we mean that we will be able to compile the entire RTS demonstrator software stack, including mocked/simulated sensors and actuators—to a normal development platform such as Linux/x86 or macOS/(x86+ARM) and execute the hand-written and generated tests therein.

2.2.3. Opportunistic Rigorous Validation and Verification

If project resources remain after accomplishing the above V&V, and if the NRC is interested in seeing further evidence of the power and efficacy of modern RDE, we also intend to demonstrate the rigorous validation or formal verification of the implementation against its requirements by one or more of the following means.

We perform several kinds of validation and verification in our safety-critical/mission-critical secure systems engineering. First, we will **[name]** and (following the first '/') summarize these different aspects of assurance, ('/') mentioning what artifacts and technologies are necessary, ('/') summarizing what question the assurance answers, and ('/') what the assurance means and its impact. Occasionally there is specific mention of the artifacts relevant to a deployable NRC system akin to the Digital Instrumentation and Control Systems system that is the focus of this project.

Assurance arguments sometimes come with evidence, and when such is present the means by which it is provided is appended to the name of the assurance case. **[PropSMT]** means that an Satisfiability Modulo Theories (SMT) solver is used to prove properties related to **Prop**, and those SMT solvers can sometimes emit a proof which can be independently checked. **[PropProof]** means that the evidence is in the form of a mechanized mathematical proof which can be independently checked. **[PropRV]** means that the evidence is in the form of a rigorous test suite against which an implementation is checked by runtime verification—the implementation is executed on the platform and every test in the test suite is checked as an assertion at runtime. **[PropMC]** means that the evidence is in the form of a rigorous check of model properties using some form (symbolic, explicit state, etc.) of model checking.

- **[RequirementRefinementToImpl]** / completeness of the implementation against its requirements by demonstrating a refinement chain starting from the **Lando** specification and ending in the verified implementations of each event in every scenario / **[Is the implementation complete with respect to its requirements?]** / ***(formal assurance using proof that an implementation conforms to its requirements)***,
- **[SafetyModelFormalProof]** / *correctness verification of actuation logic* by a formal proof of the property in the HOL domain engineering model and *in Cryptol and the verification of the implementation against the corresponding refinement invariants on the software and hardware implementations* (resp.) / **[Is this model correct and does it fulfill all safety properties?]** / ***(formal assurance using automated and interactive proof tool in the logical framework to assure that the safety model of the system is correct, and thus provides a legitimate, sound, and correct model for verification of the implementation, as discussed below)***, in particular:
  - ***[CoincidenceProof]*** / *correctness of the "two-out-of-four coincidence logic of like trip signals"*;
  - ***[FirstDeviceProof]*** / *correctness of the "logic to actuate a first device based on an OR of two instrumentation coincidence"*; and
  - ***[SecondDeviceProof]*** / *correctness of the "logic to actuate a second device based on the remaining instrumentation coincidence signal"*.
- **[ImplAssurance]** / correctness verification of every feature in the system by rigorous model-based validation and formal verification of their implementations of their implementations against their specifications / **[What evidence, rigorous and/or formal, exists, reviewable by any third party, that the (hardware/firmware/software/systems) implementation fulfills its specifications?]** / ***(perform rigorous validation/testing/runtime verification and formal verification, sometimes with proof, that a particular relation holds between specification and hardware/firmware/software/systems implementations, either/both in source and/or binary forms)***, in particular:
  - **[SpecsTypeCorrect]** / *well-formedness of the* HOL, ACSL, *Cryptol, SVA, and BlueCheck specifications through type checking*;

- **[SpecsConsistentSMT]** / *consistency of the* HOL, ACSL, and *Cryptol specifications through rigorous model-based test generation of false/bottom theorems/properties/implementations*;
- **[SoftwareSourceImplCorrectAndSafeSMT]** / correctness of C implementations against their ACSL specification using Frama-C's EVA (to prove the absence of undefined behavior causing crashes), RTE (to prove the absence of runtime errors that can cause crashes), and WP (full functional correctness) plugins;
- **[ActuationSpecCorrectSMT]** / *correctness of the Cryptol specification of the actuation algorithm by formal verification of Cryptol properties using the Cryptol tool and automated SAT/SMT solving*;
- **[SoftwareImplCorrectandSafeSMT]** / correctness of the C implementations against their Cryptol specification by writing SAWscript proof scripts and using the SAW tool to prove that the implementation corresponds perfectly to its specification;
- **[SoftwareBinaryImplCorrectAndSafeACSLRV]** / correctness of the RISC-V binaries against their ACSL specification via rigorous validation using trace-based runtime verification against executable models of the ACSL generated by INRIA's StaDy tool or similar;
- **[SoftwareBinaryImplCryptolRV]** / correctness of the RISC-V binaries against their Cryptol specification via rigorous validation using trace-based runtime verification against model-based tests generated by the Cryptol tool;
- **[SoftwareBinaryAndSourceConform]** / correctness of the RISC-V binaries against the C source by using only the CompCert compiler to generate the binaries;
- **[SoftwareBinaryImplCorrectSMT]** / correctness of the RISC-V binaries against their Cryptol specification via formal verification using the SAW tool;
- **[SoftwareImplCorrectMC]** / correctness of the C source against their behavioral specifications as expressed through inline assertions using CBMC;
- **[HardwareSourceCorrectMC]** / correctness of the SV source against their behavioral specifications as expressed through inline assertions using EBMC;
- **[BSVHardwareSourceCorrectRV]** / correctness of BSV implementation against its BSV specification using rigorous validation with BlueCheck;
- **[BSVHardwareSourceCorrectSMT]** / correctness of BSV implementation against its Cryptol specification using SAW;
- **[BSVHardwareSourceCorrectProof]** / correctness of BSV implementation against its HOL specification using Kami, if the HOL specification was written in Coq;
- **[SVHardwareSourceCorrectRV]** / correctness of the SV implementation against its UVM and SVA specifications using property-based runtime verification in simulation via software and FPGA-based simulators;
- **[SVHardwareSourceCorrectSMTMC]** / correctness of the SV implementation against its SVA specification using Yosys, Siemens EDA's Questa Formal Verification App, and Stanford's UPSCALE tool suite;
- **[SVCPUHardwareSourceCorrectRV]** / correctness of the RISC-V CPU against the RISC-V ISA specification by rigorous validation using runtime verification of the CPU, running either/both in software simulation or on the FPGA, against the RISC-V ISA compliance test suite;
- **[SVCPUHardwareSourceCorrectSMT]** / correctness of the RISC-V CPU against the RISC-V ISA specification by formal verification using Yosys and OneSpin's RISC-V Integrity Verification Solution;
- **[SoftwareASMCorrectSMT]** / correctness checking of the demonstrator's state machine against its Cryptol-based specification using SAW or Aoraï; and
- **[SoftwareCorrectRV]** / correctness of the software components, software and hardware subsystems, and overall system by rigorous validation using runtime verification via:

∗ **[SoftwareComponentsCorrectRV]** / model-based generation of unit tests for C functions and sequences of functions using CEA's Pathcrawler or INRIA's StaDy tool or similar;
∗ **[SoftwareSubsystemsCorrectRV]** / runtime verification of subsystem and system scenarios and their underlying requirements against hand-written, parametrized subsystem tests whose specifications are refinements of system scenarios expressed in HOL and Cryptol; and
∗ **[IntegratedSystemCorrectRV]** / runtime verification of software (RISC-V binaries) and hardware (SV or BSV-based bitstreams loaded onto the FPGA) against executable models by checking trace-based refinement-based simulation/emulation runs.

2.2.4. On Our Choice of Implementation Languages

We chose to implement the HARDENS demonstration system in the C programming language because (a) we expect that the NRC is familiar with C as a typical software language used by most embedded systems engineers, and (b) we use and have created many tools for synthesizing correct C code, rigorously validating C implementations, and formally verifying C source code, intermediate representations of compiled C code (e.g., LLVM [39]), and binaries generated via compilation of C source code. This does not mean that we believe C is the best programming language for use in safety-critical systems, such as those found in NPP systems.

Were we to create a high-assurance product for NPP control systems, we would advocate for an architecture that included formally verified implementations of all software subsystems in three programming languages: C (as we have explained in this proposal and will demonstrate in the HARDENS project), Rust [9], and SPARK [11]. The Rust and SPARK languages are designed for safety, avoid many of the complications and under-definedness of the C language, and we regularly use and have created comparable rigorous validation and formal verification tools for these languages as well. We are not offering to use these languages in this prototype simply because they are likely to be quite unfamiliar to the NPP.

2.3. Omitted Critical Project Aspects

We feel it is important to note that several critical project facets are not mentioned in the RFP and should be addressed in due course before moving down the path of encouraging MBE-based safety-critical systems engineering in NPPs.

In particular, any safety-critical system developed and deployed today must also attend to matters of *correctness* and *security*. The state of the art in the specification, validation, and verification of correctness and security properties in RDE—spanning hardware, software, and systems engineering—has advanced tremendously over the past decade.

Thus, we recommend that any follow-up project must include the specification, validation, and verification of a threat model, safety and cybersecurity risk analysis, and mitigation plans for both dimensions. Moreover, we believe that any modern system that sits at the center, and at the outer interfaces, of a 21st century NPP must have formally verified and rigorously validated hardware, software, and firmware at its core. Hand-written and manually tested software running on commodity hardware and operating systems is a disaster waiting to happen. Those at the forefront of RDE can do *enormously* better than that, and the NRC should demand such in (at least) future NPP Digital I&C systems.

Another potential avenue for future work is the use of modern, high-assurance, security-centric ASICs rather than commodity CPUs or FPGAs in NPP Digital Instrumentation and Control Systems systems. New security-centric SoCs are becoming available and are capable of running high-assurance operating systems like seL4 [37], INTEGRITY [28], and BedRock [5] and commodity operating systems like VxWorks [47], Linux, and FreeBSD. We include in this class of devices new RISC-V SoCs from SiFive, new ARM SoCs that include capabilities-centric security IP such as CHERI [2, 49], and technologies like Dover Technology's CoreGuard [17], which focuses on ensuring that software misbehavior caused by errors or adversaries cannot compromise a system's safe behavior.

In summary, there is an enormous opportunity to ensure that digital systems supporting NPPs of the future are safe, correct, and secure, and given technology advancements in applied formal methods and RDE over the past two decades, there is no reason not to demand that the best technology with the strongest possible assurance is used.

In the next section we detail a Statement of Work and its complementary Project Plan we offer to the NRC for this first demonstrator project.

3. Statement of Work

The HARDENS project consists of five tasks conforming to those stipulated by the RFP: **1.1 Implementation**, **1.2 Validation and Verification**, **1.3 Evaluation**, **1.4 Presentation**, and **1.5 Project Management and Final Report**. These tasks are performed over a period of eight months. Fairly granular milestones are provided for core salient checkpoints with estimated completion weeks. Each task but for the final project management task has at least one deliverable to the client. Note that Tasks 1.1 and 1.2 overlap by one month.

Task	Description	Period of Performance							
		1	2	3	4	5	6	7	8
1	Implementation	▨	▨	▨	▨				
2	Validation and Verification				▨	▨			
3	Evaluation						▨		
4	Presenation							▨	
5	Project Management and Final Report	▨	▨	▨	▨	▨	▨	▨	▨
	Milestones		*M1*		*M2*	*M3*	*M4*	*M5*	*M6*

Milestone		Month
M1	Specifications complete	2
M2	Implementation Complete	4
M3	Validation and Verification Complete	5
M4	Evaluation Complete	6
M5	Presentation Complete	7
M6	Final Report Complete	8

Figure 8: The HARDENS Project's Gantt Chart

[1.1] Implementation (duration: 4 months)

Approach: Galois will implement the system described above using both (1) highly integrated computer-based engineering development processes and (2) model-based systems engineering. All the modules of the simple protection system will be modeled functionally, and one FPGA-based circuit card will be modeled/designed in detail. The deliverable will be the model-based design itself. We will use Galois's RDE process and methodology to achieve this goal, as well as the V&V in Task 2.

The implementation's core features are those specified in the RFP and discussed in some detail in Section 2. We intend to include the following subsystems in the architecture:

- a low-cost, consumer, COTS FPGA developer board that is well-supported by commercial and open source hardware development tools;
- multiple high-assurance Soft-core RISC-V CPUs running on the FPGA, on which the core algorithm and instrumentation units run as a set of decoupled bare-metal RISC-V executables (with no operating system in use);
- multiple high-assurance hardware-only instrumentation units running on the FPGA;
- a basic text-based UI with a UX matching that specified in the RFP, implemented in software and running on a RISC-V CPU;
- pressure and temperature instruments realized as both virtual devices and physical devices (very low cost consumer sensors from Adafruit), connected to the FPGA board via GPIO;
- demonstration actuators realized as both virtual devices and physical devices (very low cost consumer push-pull solenoids from Adafruit), connected to the FPGA board via GPIO; and
- partitioning of the two trains of actuation logic across the software/hardware boundary.

Milestone/Deliverables: **[M1]** Specifications Complete (month 2) **[M2]** Implementation Complete (month 4) **[D1]** Model-based specifications complete and delivered to client (month 3) **[D2]** Implementation complete and delivered to client (month 4)

[1.2] Validation and Verification (duration: 2 months)

Approach: Galois will perform preliminary V&V and testing of the design using model-based engineering and testing methods. The deliverable will be the artifacts as described in the proposal.

Milestone/Deliverables: **[M3]** Validation and Verification Complete (month 5) **[D3]** Assurance evidence complete and delivered to client (month 5)

[1.3] Evaluation (duration: 1 month)

Approach: Galois will participate in an evaluation of the artifacts produced in tasks 1 and 2 with NRC staff. This will consist of:

- An initial kickoff meeting for this task with the NRC staff;
- Initial feedback from the NRC staff on the artifacts produced and additional necessary information;
- Galois work to address any issues and provide the additional information within the time for this portion of task 3 (1 month); and
- A second meeting to discuss the additional information provided.

Milestone/Deliverables: **[M4]** Evalution Complete (month 6) **[D4]** Galois response delivered to client (month 6)

[1.4] Presentation (duration: 1 month)
Approach: Galois will develop a day-long virtual presentation on the results of this research that explains the MBSE approach used, the engineered development environment, the development products, and lessons learned from interacting with the regulator. The deliverable will be the presentation and associated materials. *Milestone/Deliverables*: **[M5]** Presentation Complete (month 7) **[D5]** Galois presentation delivered to client (month 7)

[1.5] Project Management and Final Report (duration: 8 months)
Approach: Galois will manage the project and develop a final report describing the work performed and the results and conclusions derived. The final report shall include findings and recommendations for future research. *Milestone/Deliverables*: Annotated slide presentations; quarterly coordination reports; System Development Plan (SDP); source code; software documentation; **[M6]** end of project technical report (month 8).

4. Key Personnel

The Principal Investigator (PI) on this effort, Dr. Joseph Kiniry, will act as project lead, coordinating and supervising the successful technical execution of the project. Dr. Kiniry will also serve as the Technical Point of Contact for the project and will manage the budget of the project, oversee the engineering effort, and ensure that the work produced meets Galois's high standards. In addition, he will analyze the results and write the final report for the project and work closely with Galois business development personnel in the commercialization aspects of this work.

Table 1: Dr. Joseph Kiniry - Principal Investigator

Name	Dr. Joseph Kiniry
Project Role	Principal Investigator
Education	Ph.D., Computer Science, California Institute of Technology, 2002 M.S., Computer Science, California Institute of Technology, 1998 M.S., Computer Science, University of Massachusetts, Amherst, 1995 B.S., Computer Science, Florida State University, 1992 B.S., Mathematics, Florida State University, 1992
Relevant Experience	Dr. Joseph Kiniry is a Principal Scientist at Galois. Previously, he was a Full Professor at the Technical University of Denmark where he was the Head of the Software Engineering section. Since the early 2000s he has held permanent positions at four universities in Denmark, Ireland, and The Netherlands. Dr. Kiniry has extensive experience in formal methods, high-assurance software and hardware engineering, rigorous digital engineering, foundations of computer science and mathematics, and information security. Specific areas that he has worked in include software and hardware verification foundations and tools, the RISC-V ISA, digital election systems and democracies, smart-cards, smart-phones, critical systems for nation states, and CAD systems for asynchronous hardware.
Selected Publications	"The BESSPIN Tool Suite: Balancing the Evaluation of System Security Properties with Industrial Needs", with Daniel Zimmerman and Max Orhai. Proceedings of GOMACTech 2020. "A Formally Verified Cryptographic Extension to a RISC-V Processor", with Daniel Zimmerman, Robert Dockins, and Rishiyur Nikhil. Proceedings of the Second Workshop on Computer Architecture Research with RISC-V 2018. "Verified Visualisation of Textual Modelling Languages", with Fintan Fairmichael. Electronic Communications of the EASST 2010. "Agile Formality: A 'Mole' of Software Engineering Practices", with Vieri del Bianco and Dragan Stosic. Agile Methods + Formal Methods 2010. "Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines", with Mikoláš Janota and Goetz Botterweck. Lero Technical Report Lero-TR-SPL-2008-02, University of Limerick, 2008. "A Verification-centric Software Development Process for Java", with Daniel Zimmerman. The 9th International Conference on Software Quality 2009. "Ensuring Consistency between Designs, Documentation, Formal Specifications, and Implementations", with Fintan Fairmichael. The 12th International Symposium on Component Based Software Engineering 2009. "Secret Ninja Formal Methods", with Daniel Zimmerman. The 15th International Symposium on Formal Methods 2008. "Reasoning about Feature Models in Higher-Order Logic", Mikoláš Janota and Joseph Kiniry. The 11th International Software Product Lines Conference 2007. "The Specification of Dynamic Distributed Component Systems", M.S. Thesis, Caltech 1998.

5. Past Performance

Table 2: BESSPIN Project Details

Contract Name	DARPA System Security Integration Through Hardware and Firmware (SSITH)
Short Description	DARPA SSITH is investigating the integration of security hardware IP into modern ISAs, CPUs, and SoCs. Galois has been developing Digital Engineering artifacts such as requirements, architectures, formal models, and Digital Twins for SSITH platforms; infrastructure and tools for defining and reasoning about product lines and their derivable products; tools to formally reason about the correctness and security of bespoke and COTS hardware and firmware through the extraction of feature models and architectures from hardware designs, firmware/software source code, binaries, and test and verification benches; and tools to measure Size, Weight, and Power (SWaP)/Power, Performance, Area, and Security (PPAS) metrics for evidence-based trade-off analysis.
Client Point of Contact	Keith Rebello, Program Manager; keith.rebello@darpa.mil
Date of Completion	08/30/2021

Table 3: DARPA SHAVE Project Details

Contract Name	DARPA Software/Hardware Assurance Verified End-to-End (SHAVE)
Short Description	DARPA SHAVE is a formal method and associated tooling for end-to-end assurance of hardware/software systems. The goal of SHAVE is to inform about, and assess the feasibility of, a practical end-to-end assurance case for mission critical systems that run on COTS and bespoke hardware.
Client Point of Contact	Kerry Hill, Program Manager; kerry.hill@us.af.mil
Date of Completion	12/30/2017

Table 4: ElectionGuard Project Details

Contract Name	Microsoft ElectionGuard
Short Description	ElectionGuard is a cryptographic software developers kit for election technology providers. It implements a cryptographic protocol that facilitates the generation of end-to-end verifiable evidence of an election's correctness and security. Galois designed and developed ElectionGuard's first release using our RDE methodology. The system was specified in precise English, a formal model for the protocol and its software Application Programming Interface (API) was specified in the Coq logical framework, and the implementation had traceability from source to requirements through the formal model.
Client Point of Contact	Josh Benaloh, Senior Cryptographer; benaloh@microsoft.com
Date of Completion	09/30/2019

Table 5: 21CC Project Details

Contract Name	21st Century Cryptography (21CC)
Short Description	The 21CC chip is a GF12LP ASIC that includes high-performance, side-channel-resistant hardware implementations of AES and SHA. It also includes a RISC-V core and HMAC. The SV is synthesized from Cryptol specifications.
Client Point of Contact	James Wilson, Program Manager; james.wilson@darpa.mil
Date of Completion	09/30/2021

Table 6: Air Force/AFWERX GLASS-CV Project Details

Contract Name	Air Force/AFWERX GLASS-CV
Short Description	Galois Low-energy Asynchronous Secure SoC for Computer Vision (GLASS-CV) includes multiple RISC-V cores, an Hardware Security Module (HSM) that includes 21CC's crypto, various I/Os, and an advanced vision neural network accelerator. The cryptography in GLASS-CV reused that which was created in 21CC. GLASS-CV is also an asynchronous design.
Client Point of Contact	Vipul Patel, TPOC; vipul.patel@us.af.mil
Date of Completion	11/20/2020

Table 7: TYFONE Code Audit Project Details

Contract Name	TYFONE Code Audit
Short Description	Galois was tasked with analyzing the correctness and security of a multi-factor authentication device for use in the IC that included bespoke secure hardware, firmware, and software.
Client Point of Contact	Drew Thomas, TPOC; drew.thomas@tyfone.com
Date of Completion	1/31/2015

Table 8: GULPHAAC Project Details

Contract Name	Galois Ultra-Low-Power High-Assurance Asynchronous Crypto (GULPHAAC)
Short Description	The GULPHAAC chip is a 130nm ASIC that contains multiple formally synthesized (from Cryptol) high-assurance, high-performance, asynchronous AES, Simon, and Speck algorithms. The ASIC operates at threshold voltage or at standard voltage for high performance.
Client Point of Contact	Brian Weeks, Program Manager; beweeks@tycho.ncsc.mil
Date of Completion	10/31/2016

Table 9: Proving Amazon's s2n correct Project Details

Contract Name	Proving Amazon's s2n correct
Short Description	Galois collaborated with Amazon to show that this benefit extends to verifiability by proving the correctness of s2n's implementations of both the keyed-HMAC algorithm and the Deterministic Random Bit Generator (DRBG). To construct these proofs, we used Galois's SAW to show equivalence between specifications written in Galois's Cryptol language and the C code of s2n
Client Point of Contact	Byron Cook, AWS Security Automated Reasoning Group Lead; byron@amazon.com
Date of Completion	02/2019

Table 10: NHTSA Project Details

Contract Name	NHTSA Verified Message Parser for V2V Communications
Short Description	The goal of this project was to develop high-assurance, formally verified parsers and serializers (i.e., decoders and encoders) for the Basic Safety Message (BSM) of J2735. We succeeded in generating C code for the BSM as well as proving correctness of large parts of the code. Dr. Tullsen resuscitated and brought to life an in-house high-assurance ASN.1 compiler (developed by Galois for a previous DoD project) which was used to generate the C code. We developed the code correctness proof using the SAW code-verification tool.
Client Point of Contact	Art Carter, TPOC; arthur.carter@dot.gov
Date of Completion	1/26/2017

6. References

[1] *Ansys SCADE Suite*. Ansys. URL: `https://www.ansys.com/products/embedded-software/ansys-scade-suite`.

[2] *ARM Morello Program*. ARM. URL: `https://developer.arm.com/architectures/cpu-architecture/a-profile/morello`.

[3] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 2012.

[4] Martin Becker, Soeren Kemmann, and KC Shashidhar. "Integrating Software Safety and Product Line Engineering using Formal Methods: Challenges and Opportunities." In: *SPLC Workshops*. 2010, pp. 129–136.

[5] *BedRock Systems Inc.* URL: `https://bedrocksystems.com/`.

[6] Peter A Beerel. "Automatic gate-level synthesis of speed-independent circuits". In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 1992.

[7] Peter A Beerel, Georgios D Dimou, and Andrew M Lines. "Proteus: An ASIC flow for GHz asynchronous designs". In: *IEEE Design & Test of Computers* 28.5 (2011), pp. 36–51.

[8] Peter A Beerel, Recep O Ozdag, and Marcos Ferretti. *A designer's guide to asynchronous VLSI*. Cambridge University Press, 2010.

[9] Jim Blandy, Jason Orendorff, and Leonora Tindall. *Programming Rust: Fast, Safe Systems Development*. O'Reilly, 2021.

[10] Bob Carpenter. *Type-logical semantics*. MIT press, 1997.

[11] Bernard Carré and Jonathan Garnsworthy. "SPARK: An Annotated Ada Subset for Safety-critical Programming". In: *Proceedings of the conference on TRI-ADA'90*. 1990, pp. 392–402.

[12] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. "SAW: The software analysis workbench". In: *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*. 2013, pp. 15–18.

[13] Savio Chau. "High Assurance Systems Engineering". In: (1999).

[14] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. "Continuous formal verification of Amazon s2n". In: *Proceedings of CAV 2018*.

[15] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Germán Puebla, Balthasar Weitzel, and Peter YH Wong. "HATS-a formal software product line engineering methodology". In: *Proceedings of FMSPLE 2010*.

[16] Byron Cook. "Formal reasoning about the security of Amazon Web Services". In: *Proceedings of CAV 2018*.

[17] *CoreGuard*. Dover Microsystems. URL: `https://www.dovermicrosystems.com/solutions/coreguard/`.

[18] *Cryptol: The Language of Cryptography*. URL: `https://cryptol.net/`.

[19] David L Dill and John Rushby. "Acceptance of formal methods: Lessons from hardware design". In: *IEEE Computer* 29.4 (1996), pp. 23–24.

[20] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. "Constructing semantic models of programs with the software analysis workbench". In: *Proceedings of VSSTE 2016*.

[21] Aaron W Fifarek, Lucas G Wagner, Jonathan A Hoffman, Benjamin D Rodes, M Anthony Aiello, and Jennifer A Davis. "SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements". In: *NASA Formal Methods Symposium*. Springer. 2017, pp. 420–426.

[22] Dimitra Giannakopoulou, Anastasia Mavridou, Julian Rhein, Thomas Pressburger, Johann Schumann, and Nija Shi. "Formal Requirements Elicitation with FRET". In: (2020).

[23] *Gitflow Workflow*. URL: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow`.

[24] *GitLab 14*. URL: `https://about.gitlab.com/`.

[25] Dylan Hand, Matheus Trevisan Moreira, Hsin-Ho Huang, Danlei Chen, Frederico Butzke, Zhichao Li, Matheus Gibiluka, Melvin Breuer, Ney Laert Vilar Calazans, and Peter A Beerel. "Blade–a timing violation resilient asynchronous template". In: *Proceedings of ASYNC 2015*.

[26] Mats Per Erik Heimdahl and Barbara J Czerny. "Using PVS to Analyze Hierarchical State-based Requirements for Completeness and Consistency". In: *Proceedings. IEEE High-Assurance Systems Engineering Workshop (Cat. No. 96TB100076)*. IEEE. 1996, pp. 252–262.

[27] *IEEE Standard Criteria for Safety Systems for Nuclear Power Generating Stations*. Tech. rep. IEEE 603-2018.

[28] *Integrity RTOS*. Green Hills Software. URL: `https://www.ghs.com/products/rtos/integrity.html`.

[29] Mikoláš Janota, Joseph Kiniry, and Goetz Botterweck. "Formal methods in software product lines: concepts, survey, and guidelines". In: *Lero, University of Limerick, Tech. Rep. TR-SPL-2008-02* (2008).

[30] *JasperGold Formal Verification Platform*. URL: `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html`.

[31] J-M Jézéquel and Bertrand Meyer. "Design by contract: The lessons of Ariane". In: *Computer* 30.1 (1997), pp. 129–130.

[32] Joseph Kiniry and Fintan Fairmichael. "Ensuring Consistency Between Designs, Documentation, Formal Specifications, and Implementations". In: *Component-Based Software Engineering* (2009).

[33] Joseph Kiniry, Fintan Fairmichael, and Eva Darulova. *Beetlz: A BON Software Model Consistency Checker for Eclipse*. Tech. rep. University College Dublin, 2009.

[34] Joseph R Kiniry and Fintan Fairmichael. "Ensuring consistency between designs, documentation, formal specifications, and implementations". In: *Proceedings of CBSE 2009*.

[35] Joseph R Kiniry, Daniel M Zimmerman, Robert Dockins, and Rishiyur Nikhil. "A formally verified cryptographic extension to a RISC-V processor". In: *CARRV 2018*.

[36] Joseph R. Kiniry and Daniel M. Zimmerman. "Secret Ninja Formal Methods". In: *Proceedings of Formal Methods 2008*. May 2008.

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[38] Brian R Larson, Patrice Chalin, and John Hatcliff. "BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software". In: *NASA Formal Methods Symposium*. Springer. 2013, pp. 276–290.

[39] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86.

[40] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[41] Alain J Martin. *Programming in VLSI: From communicating processes to delay-insensitive circuits*. Tech. rep. Caltech, 1989.

[42] Alain J Martin and Mika Nystrom. "Asynchronous techniques for system-on-chip design". In: *Proceedings of the IEEE* 94.6 (2006), pp. 1089–1120.

[43] Carroll Morgan. "The refinement calculus". In: *Program Design Calculi*. Springer, 1993, pp. 3–52.

[44] *OneSpin Solutions*. URL: `https://www.onespin.com/`.

[45] Mark Richters and Martin Gogolla. "OCL: Syntax, Semantics, and Tools". In: *Object Modeling with the OCL*. Springer, 2002, pp. 42–68.

[46] Ina Schaefer and Reiner Hähnle. "Formal methods in software product line engineering". In: *Computer* 2 (2011), pp. 82–85.

[47] *VxWorks RTOS*. Wind River Systems. URL: `https://www.windriver.com/products/vxworks`.

[48] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. “Formal methods: Practice and experience”. In: *ACM computing surveys (CSUR)* 41.4 (2009), pp. 1–36.

[49] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. “The CHERI Capability Model: Revisiting RISC in an Age of Risk”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 457–468.

[50] Daniel M Zimmerman and Joseph R Kiniry. “A verification-centric software development process for Java”. In: *2009 Ninth International Conference on Quality Software*. 2009.