

# Rigorous Digital Engineering

Developing a Cryptographic  
Voting Protocol with RDE

**FREE & FAIR**

<https://freeandfair.us>

# Contents

Engineering a Secure Voting Protocol.....	4
An audacious request.....	4
The “Broken Telephone” Problem with Engineering.....	5
When Standard Practices Fall Short .....	5
The Cryptographic Voting Protocol .....	6
A Foundation, Not a Complete System .....	6
End-to-End Verifiable Internet Voting (E2E-VIV) .....	7
The Engineering Challenge.....	8
Introducing Rigorous Digital Engineering (RDE).....	8
The Building Blocks of Rigorous Digital Engineering .....	9
Domain Engineering: Creating a Shared Vocabulary.....	9
Requirements Engineering: Defining System Laws.....	12
Product Line Engineering: Controlled Customization.....	14
The Feature Model.....	15
Architecture: Building the Blueprint.....	16
Architecture as a verifiable blueprint .....	18
Design: From Blueprint to Specification.....	19
Design by Contract: Specifications as Guarantees.....	19
Verification-Centric Design Philosophy .....	20
Systematic Design Techniques.....	20
From Architecture to Implementation .....	20
Ready for development .....	21
Development: Specification-driven Coding.....	21
Building from the foundation up .....	21
Comprehensive Quality Assurance.....	21
Security Engineering: Withstanding Determined Adversaries .....	22
Start with Transparency .....	22
Know Your Enemy: Comprehensive Threat Modeling .....	22
Security Assurance Cases: Attack Scenarios and Mitigations.....	23
Cryptography: The Mathematical Foundation .....	24
Trust Distribution and Zero Trust Architecture .....	24
Security Throughout RDE.....	25
Verification and Validation: Proving Systems Work .....	25
Models as a Foundation for Testing.....	26

Comprehensive Verification Strategies .....	26
Validation Throughout Development.....	26
Building Genuine Confidence .....	27
Evolution and Maintenance: Preserving Trust Over Time.....	27
The Inevitability of Change .....	27
The RDE Advantage.....	28
The Long-Term Challenge: System Ownership .....	28
Applying Rigorous Engineering: The Promise and the Challenge.....	28
The Adoption Paradox.....	29
The Emerging Mainstream .....	29
Voting Protocol: What Lies Ahead.....	30
The Stakes Are Rising .....	30



## Engineering a Secure Voting Protocol

*Dan Zimmerman, Joseph Kiniry, Shpat Morina*

Hardly a month passes by without news of another disastrous software failure. Recent examples include commercial aircraft grounded worldwide due to navigation system glitches, banking networks freezing millions of transactions; healthcare systems exposing sensitive patient data to malicious actors. At first glance, it seems that trustworthy software remains elusive despite decades of progress.

The rare exceptions are critical systems designed with the understanding that failure means catastrophe; software governing nuclear reactor coolant systems, for example. Critical systems like these often undergo verification so rigorous it can take longer to demonstrate the software's correctness than to write it in the first place. And yet, even critical systems occasionally exhibit serious vulnerabilities.

Among critical systems, perhaps none carry implications as profound as those that enable democratic participation. If we cannot trust the systems that count our votes, we cannot trust the results of our elections, and democracy itself becomes vulnerable.

### An audacious request

**KEY IDEA:** *If we cannot trust the systems that count our votes, we cannot trust the results of our elections. **The goal:** Build a cryptographic protocol for online voting that can withstand the most extreme scrutiny.*

In early 2024, Tusk Philanthropies, a non-profit organization with a longstanding interest in increasing voter participation, approached our team at Free & Fair with a request: develop a

cryptographic protocol for a secure online voting platform that could withstand the most extreme scrutiny, and make it demonstrably secure.

This wasn't a typical engineering task. Online voting systems operate in an exceptionally hostile environment. They are exposed to the world and must withstand sophisticated attacks by diverse adversaries, from individual hackers to nation-states with virtually unlimited resources.

In online voting systems, every component must be unquestionably reliable and secure. We weren't being asked to build an entire voting system, but instead the most critical core component: the cryptographic protocol that sits at the center of any secure online voting platform, the mathematical foundation that makes secure, verifiable elections possible.

How do you build something that must be fundamentally trustworthy? How do you develop a system so robust that even the most determined adversaries can't effectively attack? And how do you prove that your system actually delivers this level of security?

For us, the answer is *rigorous digital engineering* (RDE), a methodology that fundamentally changes how critical systems are built and verified. Below, we explain what RDE entails and how we're using it to create a rigorously engineered cryptographic voting protocol.

## The “Broken Telephone” Problem with Engineering

The cause of countless software failures is what we might call a “broken telephone” problem. In the children's game of “telephone,” a message is whispered from person to person, becoming increasingly distorted until the final version bears little resemblance to the original. Engineering complex systems suffers from a similar phenomenon.

**KEY IDEA:** *In complex systems, every hand-off distorts the original intent. The gaps aren't about negligence; they're baked into informal translations between stages.*

In engineering, at the start there's a vision, a concept of what a system should do. This concept passes through many hands: business analysts, product managers, architects, developers, and testers. At each transition, the vision is translated, interpreted, and inevitably transformed. By the time it reaches implementation, key aspects of the original intent have often been lost, misinterpreted, or fundamentally changed.

This “broken telephone” problem is not necessarily the result of incompetence or negligence. It's a systemic issue with how complex systems are traditionally built. Requirements get misinterpreted. Hidden assumptions creep in. Security considerations get lost in translation. The gap between what was intended and what gets built widens with each handoff.

## When Standard Practices Fall Short

Modern development teams employ a range of methodologies designed to improve quality and reduce defects: agile development for flexibility, test-driven development for code quality, continuous integration for early problem detection, code reviews for human oversight, and static analysis tools to find potential problems before code executes. Most consumer applications and many business-critical systems are built using these standard practices, and they function reliably enough.

But these development approaches fundamentally rely on informal translations between stages of development. Requirements documents written in English are interpreted by humans and translated into designs, which are again interpreted and translated into code. Testing demonstrates that the final product appears to behave correctly, but doesn't necessarily show that it will behave correctly in all circumstances.

**KEY IDEA:** *Testing shows that software appears correct — not that it is correct in every circumstance. For voting, 'good enough' leaves an intolerable margin of error.*

For systems where failure isn't an option, like voting systems, these standard practices leave an uncomfortable margin of error. They don't provide the level of certainty needed to declare a system reliable and trustworthy. What's needed is something that *unambiguously* connects the beginning to the end. This is where rigorous digital engineering enters the picture.

Before diving into how rigorous digital engineering makes such a system possible, it's worth understanding exactly what we're building and why it matters.

## The Cryptographic Voting Protocol

A *cryptographic voting protocol* is the mathematical and computational foundation that enables secure, verifiable elections to be conducted over the Internet. It defines the rules of the game, and is the engine that enforces them. The protocol uses mathematics to enable remote elections that are completely transparent (anyone can audit the entire process and verify the tally) and absolutely private (no one can link a ballot to a voter or see their choices): voters can cast encrypted ballots from anywhere, and the system ensures that votes remain secret, that they cannot be altered, and that their integrity can be independently verified by anyone.

This engine has to be unquestionably reliable in any digital voting system. Otherwise, the reliability of the rest of the system is irrelevant.

### A Foundation, Not a Complete System

**KEY IDEA:** *A protocol is both the rulebook and the engine, using mathematics to guarantee privacy and transparency. It is the digital vault around which secure voting systems are built.*

Our Cryptographic Voting Protocol is not itself a complete voting system. Rather, it's the secure foundation upon which others can build full-featured voting platforms. It can be thought of as the digital equivalent of a secure vault around which an entire bank building is planned and built. The protocol defines and enforces how ballots are encrypted, how votes are anonymously shuffled and counted, how proofs of correctness are demonstrated, and how the entire process can be audited. However, it doesn't specify user interface designs or handle the countless practical details needed for elections.

This foundational approach is intentional. Our aim is to enable innovation in voting system design by creating a secure, open-source protocol that others can build upon, ensuring that the security-critical core remains consistent and verifiable. Election technology vendors, government agencies, and civic organizations can each create their own user experiences and administrative tools and

share the same mathematically secure foundation.

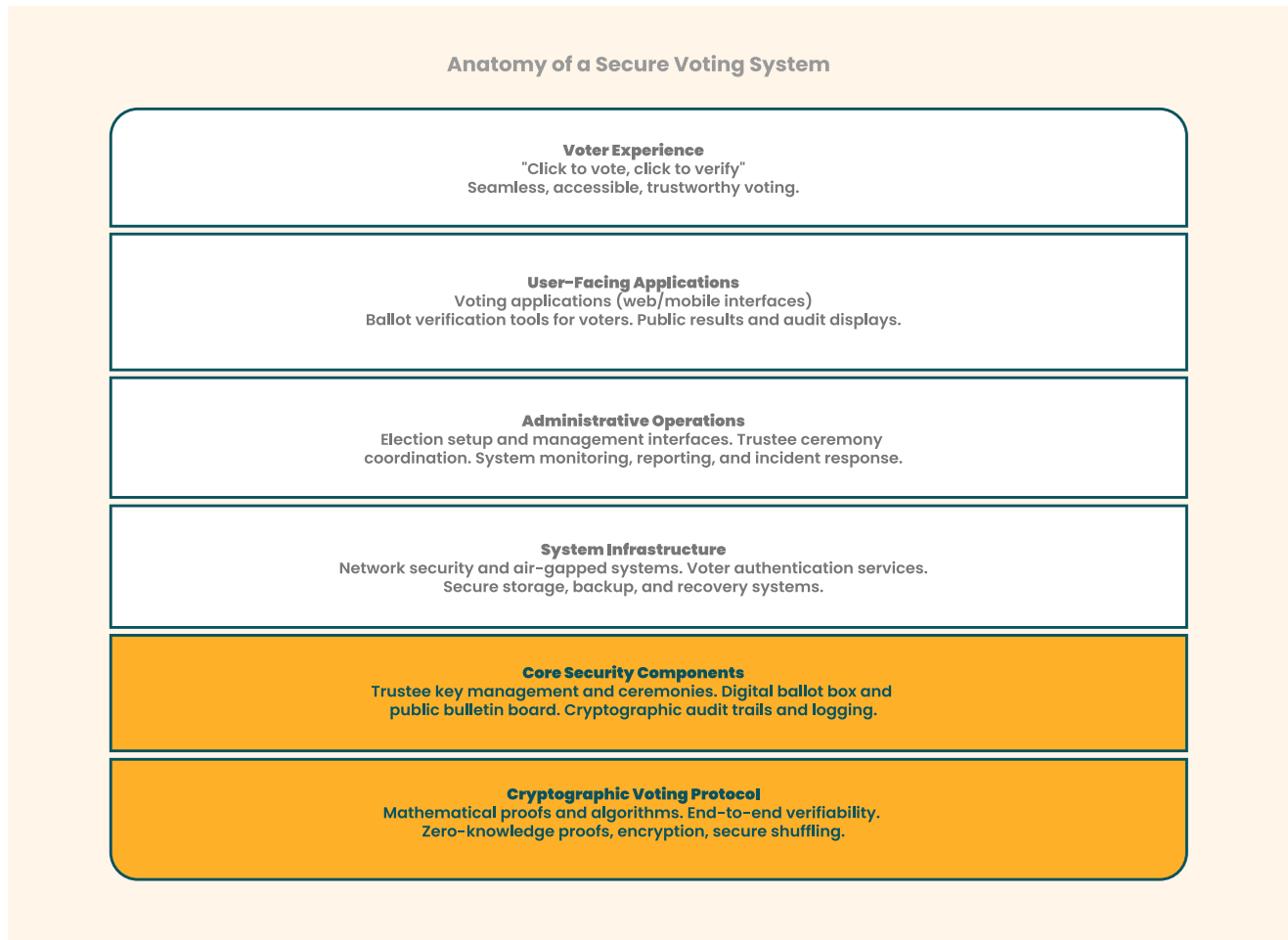
### End-to-End Verifiable Internet Voting (E2E-VIV)

The protocol implements what cryptographers call End-to-End Verifiable Internet Voting, or E2E-VIV: a voting system where every step of the process can be mathematically verified without compromising voter privacy.

In an E2E-VIV system, verification happens at three crucial levels.

1. Each voter can verify that their ballot was cast as intended—that the encrypted version matches their actual choices.
2. Each voter can verify that their ballot was recorded as cast—that the encrypted version appears correctly on a public bulletin board.
3. Anyone can verify that all recorded ballots were counted as recorded—that the final tally correctly includes every legitimate vote.

This creates what security experts call “software independence”: even if the voting software built around the protocol contains errors or has been maliciously altered, those problems cannot cause an undetectable change or error in an election outcome.



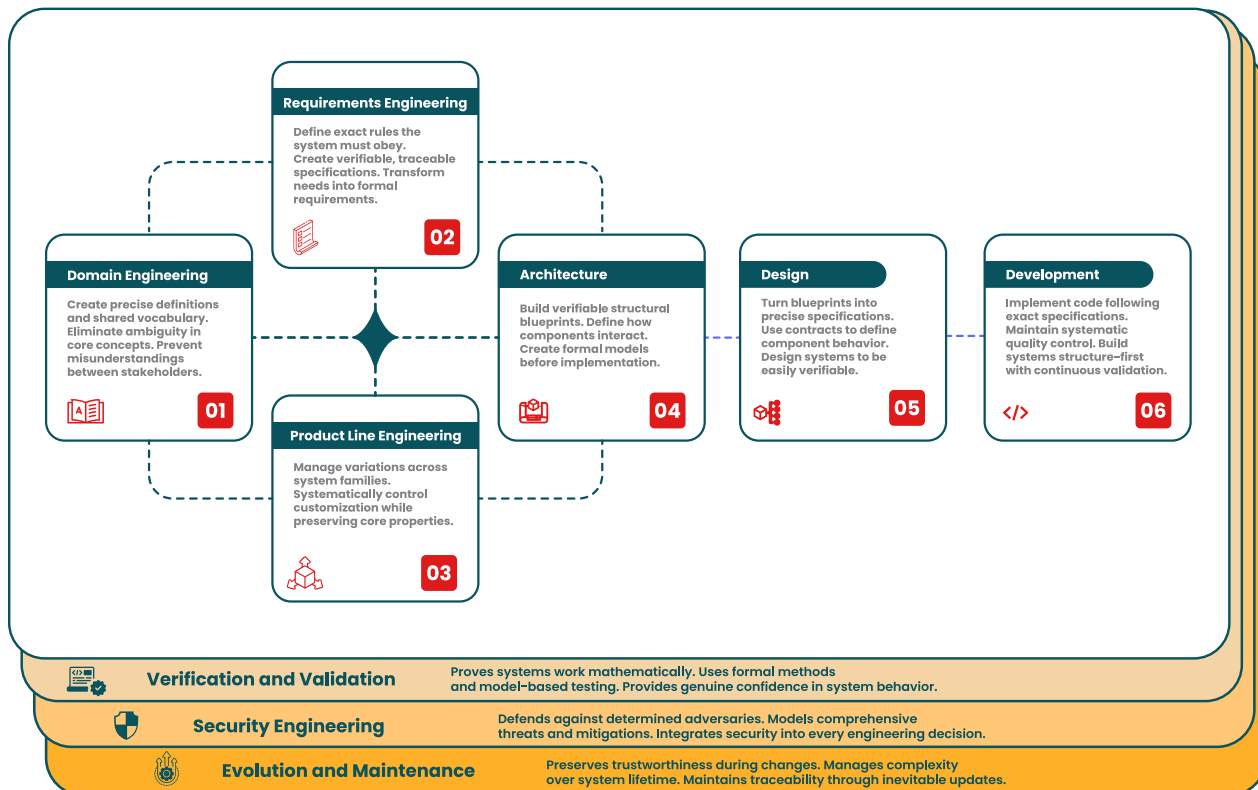
## The Engineering Challenge

**KEY IDEA:** *Protect privacy, enable public audits, resist insiders and nation-states, at the scale of millions of voters.*

Creating such a protocol presents extraordinary engineering challenges. The system must be secure against all potential malicious actors, from corrupt officials to nation-state adversaries. It must be usable by ordinary citizens, and transparent enough for public verification. It must protect voter privacy while enabling comprehensive auditing, remain secure even when some components are compromised, and maintain these properties across potentially millions of voters.

These requirements demand the highest levels of engineering rigor. The stakes are too high for traditional development approaches.

## Introducing Rigorous Digital Engineering (RDE)



*Rigorous digital engineering (RDE)* is a fundamentally different systems development approach that aims to eliminate the translation errors of the broken telephone game by introducing consistent rigor to software engineering at every step.

RDE is essentially a model-driven methodology: instead of relying on informal translations between stages of development, RDE creates precise models at each stage that can be tested, verified, and analyzed long before any traditional code is written. Further, these models are connected through a concept called refinement to create an unbroken chain from high-level concepts down to implementation details, ensuring that we add the necessary context to develop the system, without losing anything in translation.

## RDE Origins

Rigorous digital engineering is not fundamentally new, but rather a systematic combination of established methodologies with roots dating back to the 1960s. RDE draws from the academic field of “formal methods” and related mathematical techniques originally developed to prove that software behaves correctly. Until recently, these powerful approaches have remained largely confined to research laboratories because they required specialized mathematical expertise that few practicing engineers possessed.

**KEY IDEA:** *RDE replaces ambiguous prose and good intentions with rigorous development and mathematical proof.*

RDE represents a mature integration of these proven, publicly available techniques into a practical engineering framework. Rather than proprietary methods, it systematically combines well-established, user-friendly approaches that hide the mathematical complexity behind user-friendly tools, making them accessible to real-world engineering teams while preserving underlying rigor.

This coordinated methodology took shape in the mid-1990s through intensive work on safety-critical systems where failure meant catastrophic consequences. Over the last decade, flavors of RDE have been developed and refined at Galois, a defense research services company, where RDE has been applied to dozens of projects creating secure, high-assurance software and hardware systems. For example, as part of a project with the Nuclear Regulatory Commission, Galois has demonstrated RDE in developing a reactor trip safety system for nuclear power plants. These techniques have also been successfully used in aviation, financial systems, and medical devices.

Since the mid-1990s, these methodologies have adapted to new programming languages, platforms, and domains. What began as specialized application to critical systems has expanded to broader systems development, demonstrating that the same proven methods preventing disasters in nuclear reactors can be applied to many engineering challenges.

## The Building Blocks of Rigorous Digital Engineering

To understand how RDE changes the way systems are developed, let’s look at its core building blocks and how they work together to ensure reliability and security.

### Domain Engineering: Creating a Shared Vocabulary

Before we can build a reliable system, we need to first answer a deceptively simple question: What exactly are we talking about? Enter *domain engineering*, the first critical building block of rigorous digital engineering.

Domain engineering tackles a fundamental source of the “broken telephone” problem: ambiguity in the language we use to describe the system’s world. Consider our voting protocol: What does the term “ballot” refer to? Is it the screen display a voter sees, the bundle of encrypted data sent to a server, or the eventually tallied artifact? Similarly, what does “cast” mean? The click of a button, server receipt of a voted ballot, or final inclusion in the count?

**KEY IDEA:** *Even the best techniques fail if they tackle a misunderstood problem.*

In critical systems, such nuances aren't academic; they are potential failure points. These nuances might be resolved differently by different teams during development. Those discrepancies can create hidden inconsistencies that cause different parts of the system to operate under different assumptions. It's these inconsistencies that can cause vulnerabilities and unintended behavior.

This challenge becomes exponentially more complex when multiple organizations collaborate on a single project. In elections technology, these stakeholders might include administering jurisdictions, elections vendors leveraging the voting protocol to build a full fledged voting system, and organizations focused on verifying functionality. Each additional stakeholder brings their own interpretations and assumptions about core terminology.

Domain engineering forces a deliberate pause *before* solution design begins. It recognizes that even the most sophisticated engineering techniques will fail if applied to a misunderstood problem. We therefore need to rigorously identify and describe the core concepts inherent to the problem area, the "domain", and how those concepts overlap and intertwine. This process catches fundamental misunderstandings at the earliest, cheapest stage. More importantly, it provides an accurate, agreed-upon vocabulary used to build the system from that point forward.

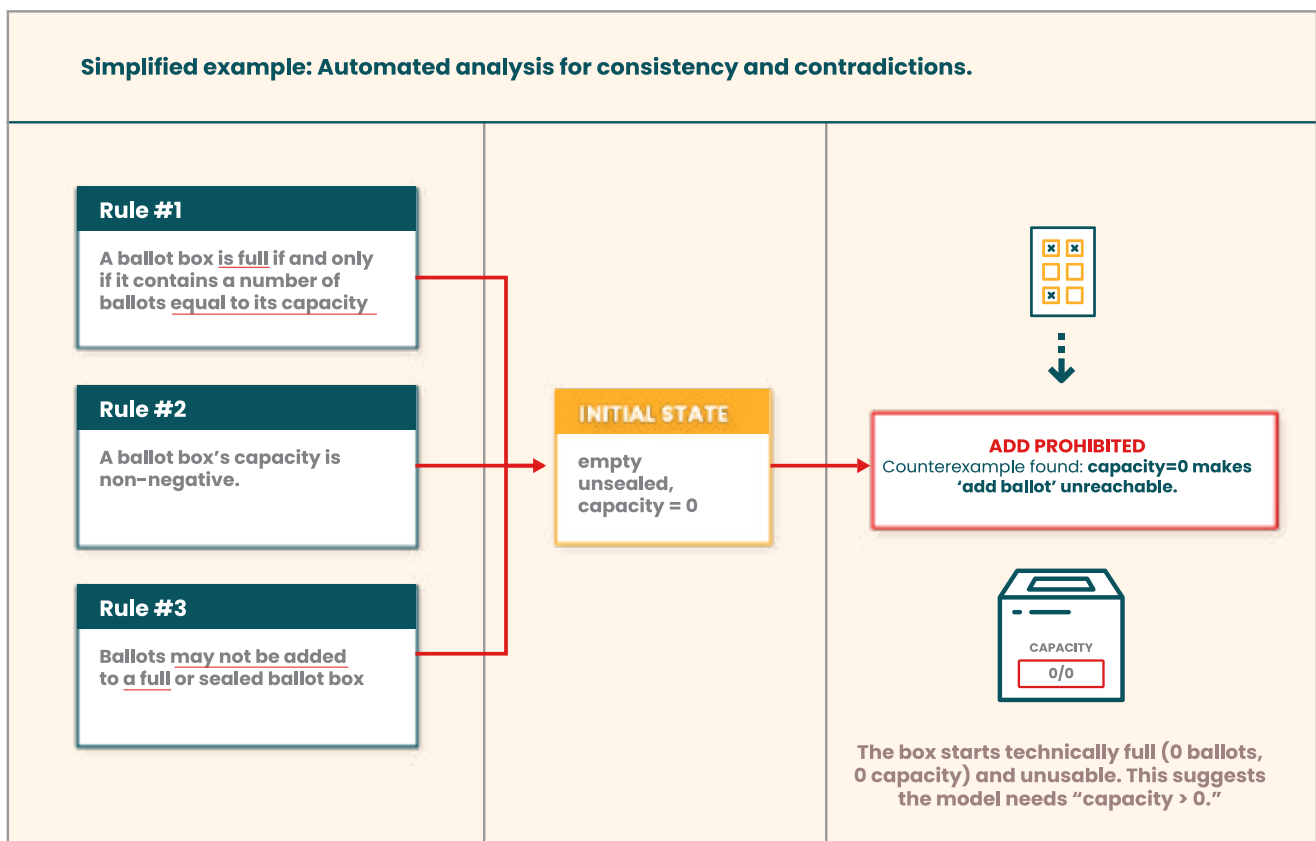
The Domain Engineering process typically includes these steps:

- **Scope definition:** *Defining the precise boundaries of the domain.* What concepts are essential to the core logic of our system, and what lies outside or is assumed background knowledge? For the voting protocol, things like the nature of a *Digital Ballot*, the function of the *Digital Ballot Box*, the process of making *Contest Selections*, and the role of *Public Key Cryptography* are definitely in scope. Conversely, anything related to voter registration databases, for example, is not crucial for defining the core cryptographic protocol logic, so it is considered to be out of scope for the domain model.
- **Concept identification & description:** *Identifying the key "things", both tangible and abstract, within the defined scope.* What are the entities, actions, events, and behaviors that exist within that scope? How is each one described, what attributes does it have, and what must hold true for it? To illustrate with a key example from our voting protocol: a *ballot* is strictly defined as the "presentation of the contest options for a particular voter." This distinguishes it clearly from related, but distinct, concepts like a *Marked Ballot* (which contains the voter's selections), a *Cast Ballot* (which represents the voter's irrevocably confirmed intent), a *Spoiled Ballot* (a ballot on which the voter may have made selections, which will not be counted because it has been explicitly invalidated), and the specific digital (*Digital Ballot*) or physical (*Paper Ballot*) manifestations. Similarly, to *Cast* is an event transitioning a *Ballot* to a *Cast Ballot* state, invoking constraints like "A *ballot* cannot be both *cast* and *spoiled*." This might seem like self-evident busywork, but nailing these definitions prevents components from operating on mismatched assumptions about fundamental entities and actions.
- **Relationship Modeling:** *Defining how concepts interconnect.* How do the identified concepts interact and depend on one another? This involves identifying several kinds of relationships:
  - **specialization** (is-a), where one concept is a specific kind of another: for example, *Digital Ballot*, which refers to a ballot comprised of one or more digital artifacts, is a type of *Ballot*, which refers to the presentation of the contest options for a particular voter.

- **containment** (has-a), often expressed through structure or constraints: a *Ballot* contains *contests*, for example.
- **associations** or usage relationships, where neither concept contains the other, but they work together: a *Voter* uses a *Voting Application* to make their selections; an *election system* uses concepts like *voter registration*, *tabulation devices*, etc.

Documenting these relationships rigorously prevents misunderstandings about how different parts of the domain influence each other.

The descriptions of what these concepts entail often start in Controlled Natural Language (CNL), a subset of English designed to minimize ambiguity. Crucially, as we'll come to see throughout this process, RDE encourages pushing towards formalization, especially for critical concepts.



This involves translating CNL into precise mathematical and logical notations. This formalization allows for automated analysis: tools that can check the domain model itself for internal consistency (absence of contradictions) and potentially explore its logical consequences, revealing hidden assumptions or edge cases *before a single requirement for the system is written*.

## Controlled Natural Language: Examples

### Ballot

- Definition: Presentation of the contest options for a particular voter.
- Queries:           What is your ballot style? Have you been cast? Have you been spoiled?
- Commands:    You are cast! You are spoiled!
  
- Constraints:
  - A ballot cannot be both cast and spoiled.
  - Once a ballot is cast, it remains cast forever.
  - A ballot may only contain votes for contests that are part of its ballot style

### Seal

- Definition: A seal provides some evidence of the integrity of another entity (such as a ballot or ballot box).
- Queries:    Are you applied? What are you applied to? Are you broken?
- Commands:   You are applied to the entity! Break!
  
- Constraints:
  - A seal may only be applied if it has not previously been applied or broken.
  - A seal may only be broken if it has not previously been broken.
  - Once applied, a seal remains applied forever.

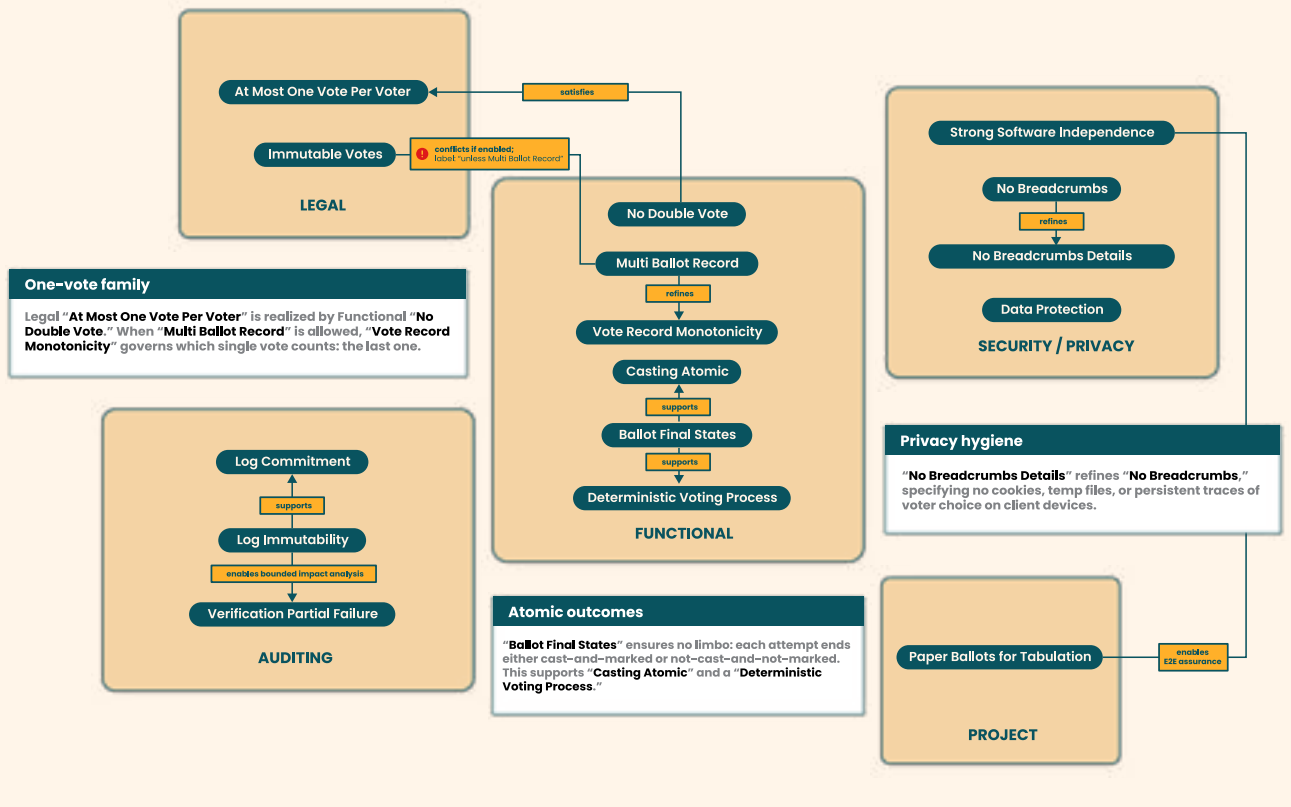
The result is a structured model—the *domain model*—that captures the essential reality of the domain.

This foundational work might seem straightforward, but it's the linchpin of RDE. The domain model establishes an unambiguous vocabulary that all stakeholders (cryptographers, architects, developers, verifiers...) must use. Everything else is anchored in the domain model: requirements engineering will exclusively use the vocabulary from the domain model; architectural components will map back to domain concepts; and verification activities will check system properties expressed in terms grounded in this shared understanding. In other words, the domain model becomes the semantic foundation for development from here on out.

## Requirements Engineering: Defining System Laws

After establishing the domain's core vocabulary, the next step in rigorous digital engineering is precisely defining what the system must *do*. This is *requirements engineering*, which establishes the system's foundational laws: the explicit, verifiable rules it must obey.

Requirements: A Structured Network of Laws



In conventional software development, requirements gathering typically manifests as collecting stakeholder requirements, documenting them in English, and hoping developers interpret them consistently, in the way they were intended, throughout the development process. In RDE, we aim to elevate this process to one that avoids the same telephone game problem we've already identified. Requirements engineering is a rigorous modeling activity that systematically defines what a system must be able to do as a precise specification.

**KEY IDEA:** *RDE turns requirements into laws the system must provably obey.*

For the voting protocol, requirements engineering began with a comprehensive analysis of several sources: the OSET Institute's prior work, U.S. Vote Foundation's "Future of Voting" report, international cryptographic voting standards, and threat considerations from election security experts. But simply collecting these requirements is just the start.

Requirements Engineering is systematized in a few distinct ways:

- Structure:** As requirements take shape, they form a structured network rather than a flat list. They are organized, often hierarchically, explicitly reflecting relationships to other requirements. This allows for reasoning about groups of requirements. For example, a high level requirement for "End-to-end verifiability" might be decomposed into several lower-level requirements detailing specific cryptographic checks voters and auditors must be able to perform.

- **Traceability:** Requirements are traced *backward* to their originating source: a stakeholder need, regulatory mandate, domain concept, or threat model (which we discuss in security engineering further down). This ensures they are grounded. More critically, they are also traced *forward* through architectural components, design elements, code implementations that realize that behavior and the verification activities that provide evidence of conformance. These verifiable threads are vital for managing complexity. They are rigorously maintained connections that allow us to analyze the impact of changes down the line, and build a coherent assurance case demonstrating how the system fulfills its objectives.
- **Increasing formality:** Just as with domain engineering, the pursuit of precision naturally leads to increasing formality: expressing requirements in more precise notations. This serves a dual purpose. First, it drastically reduces ambiguity and forces clarity of thought. Second, and perhaps more importantly, the more formal and structured the requirements, the more amenable they are to automated analysis. Automated tools can check requirements for consistency (no contradictions), completeness (no gaps), and realizability (actually possible to implement).
- **Formal mathematical logic:** For safety-critical aspects, formal requirements enable mathematical proofs of correctness. Certain core requirements addressing cryptographic integrity in a voting protocol, for example, demand the highest level of rigor. In these cases, the requirements are specified as formal mathematical logic. These might be expressed in specialized programming languages like Cryptol, designed for specifying cryptographic algorithms, or within the frameworks of interactive theorem provers, which allow for machine-checked mathematical proof of their properties. As we'll see when we discuss verification and validation, this highest level of formality permits a degree of verification far beyond traditional testing, offering mathematical certainty about the foundational security mechanisms.

The outcome of requirements engineering in RDE is a specification that is precise, structured, traceable and analyzable. This clarity becomes the foundation of everything that follows.

## Product Line Engineering: Controlled Customization

Having established a rigorous vocabulary and a precise set of rules the system must obey, we confront another source of potential chaos in complex system development: *variability*.

Complex systems are rarely monolithic: they often exist as families of related products, tailored for different contexts, regulations, and deployment configurations. For our voting protocol, for example, different jurisdictions have varying requirements for how voting should work. Some might prioritize certain accessibility features, others might have specific audit requirements, and still others might operate under unique regulatory frameworks. Rather than creating separate protocols for each scenario, we need a way to build a coherent family of protocols that share foundational security and reliability properties while accommodating these differences.

**KEY IDEA:** *Variation is inevitable; RDE manages it without sacrificing security.*

To manage this complexity, we turn to *product line engineering*: the RDE building block dedicated to the systematic management of variability across a family of systems. Where domain engineering established our conceptual foundation and requirements engineering specified what the system

must do, product line engineering (PLE) provides the framework for managing how these requirements are fulfilled across different contexts and configurations.

### **The Feature Model**

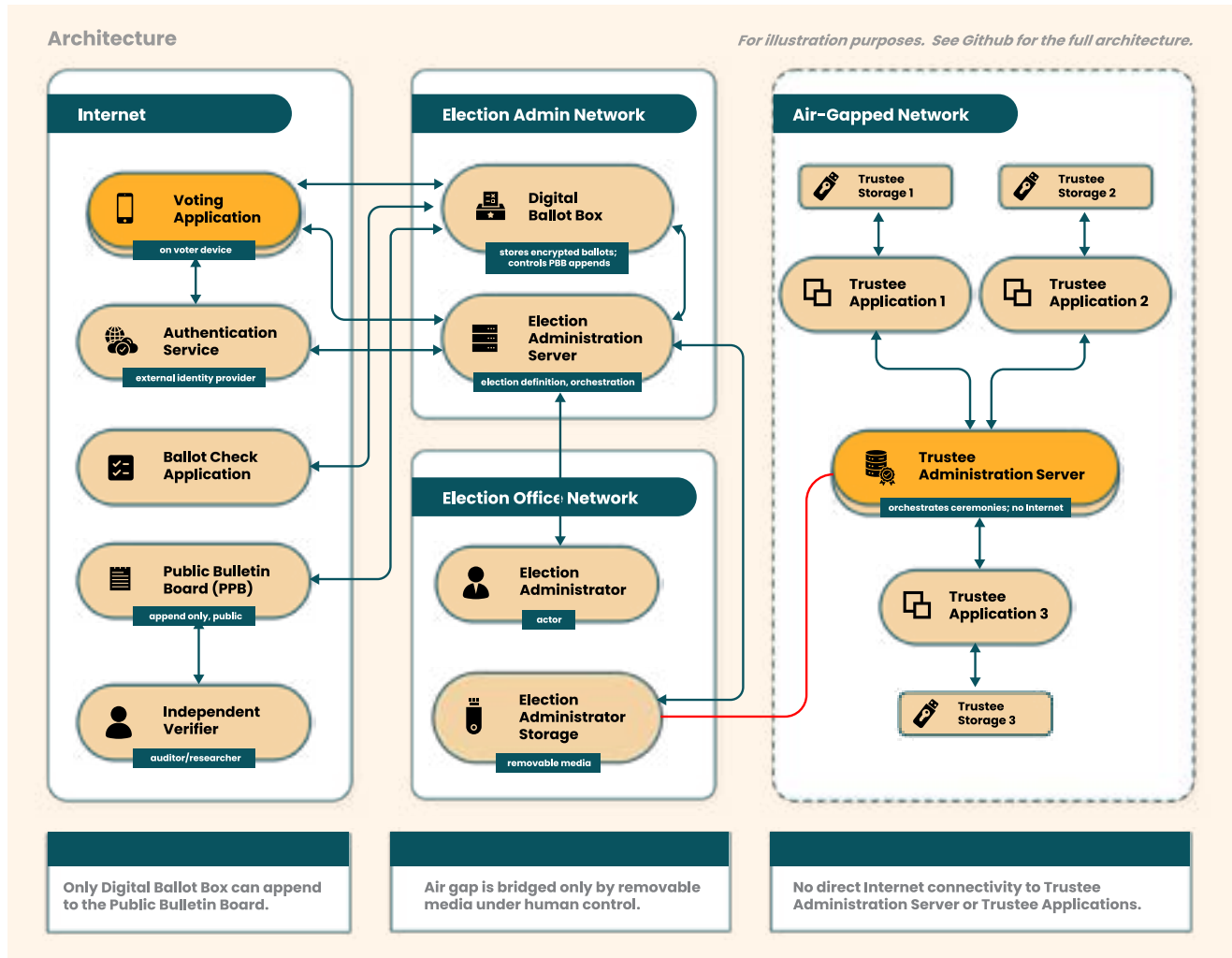
Central to PLE is the feature model, a representation of the choices available across the product line. Let's take a simplified example from our protocol: we might model variations such as the type of encryption algorithm used, whether the system supports revoting (the casting of multiple votes with only the last one counting), or what kind of receipt is provided to voters. Each of these represents a point of legitimate variation in the system. The feature model captures both the available options and the constraints among them (for instance, that certain receipt types are only compatible with specific encryption approaches).

RDE also encourages Product Line Engineering to move from informal descriptions of variability to precise, structured, and analyzable models. For each feature, we also formally capture its interdependencies, noting, for example, which one feature requires another in order to be activated, and which features are mutually exclusive. Modeling variability and interdependencies allows us to represent feature models as constraint satisfaction problems, enabling automated analysis to answer questions like "Is this particular combination of features actually feasible?" and "What are all the configurations that allow multiple ballot submission while also meeting this security requirement?"

In essence, Product Line Engineering provides the rigorous mechanisms to manage the overwhelming combinatorial explosion of protocol variants. It ensures that even as we accommodate necessary variations, we do so in a controlled, consistent, and verifiable manner, preserving the end-to-end integrity that rigorous digital engineering demands. It is a crucial step in ensuring that every valid configuration derived from our common assets upholds the stringent requirements for trustworthiness.

## Architecture: Building the Blueprint

The next step in rigorous digital engineering is to define the system's fundamental organization: its architecture. Architecture is where the system begins to take shape. This building block of RDE aims to create a precise bridge between the abstract world of requirements and the concrete reality of implementation. This is where we decide how components interact, how data flows, and critically for a voting protocol, where and how cryptographic operations occur.



For our cryptographic voting protocol, this involves decomposing the system into interconnected components and defining how all components interact, establishing communication pathways and interfaces. These components can be logical and physical building blocks like the digital ballot box responsible for receiving and storing encrypted votes. The latter needs careful consideration about how and when it communicates with the rest of the system, so we define all possible interactions at a high level; for example, the digital ballot box is the only component that can append entries to the public bulletin board, ensuring encrypted ballots and spoil requests follow authorized paths.

## Architecture: Authorized Flows

### On the Internet

- Authentication Service <-> Voting Application
  - AS -> VA: Initiates the voter's authentication prompt inside the app.
- Independent Verifier <-> Public Bulletin Board
  - Verifier -> PBB: Requests election data for audit.
  - PBB -> Verifier: Returns selected or full election data.

### Between the Internet and the Election Administration Network

- Voting Application <-> Digital Ballot Box
  - VA -> DBB: Submits encrypted ballot; requests ballot check; casts ballot; requests spoil/decryption.
  - DBB -> VA: Sends initial receipt and ballot-check information.
- Ballot Check Application <-> Digital Ballot Box
  - BCA -> DBB: Requests ballot information for checking.
  - DBB -> BCA: Returns ballot-check information.
- Election Administration Server <-> Voting Application
  - VA -> EAS: Requests authentication; signals "voter has voted."
  - EAS -> VA: Sends available elections.
- Election Administration Server <-> Authentication Service
  - EAS -> AS: Requests voter authentication for a specific election.
  - AS -> EAS: Returns authentication result.
- Digital Ballot Box -> Public Bulletin Board
  - DBB -> PBB: Appends ballot cryptograms, cast/spoil requests, ballot-check info, election manifest, and results.

### Within the Election Administration Network

- Election Administration Server <-> Digital Ballot Box
  - EAS -> DBB: Publishes manifest and results; requests election data.
  - DBB -> EAS: Provides election data.

Between the Election Administration Network and the Election Office Network

- Election Administrator <-> Election Administration Server
  - Admin -> EAS: Loads manifest; gathers election data; loads results.
- Election Administration Server <-> Election Administrator Storage (removable media)
  - EAS -> Storage: Writes election manifest and election data.
  - Storage -> EAS: Provides decrypted tally for publication.

Across the Air Gap (via removable media)

- Election Administrator Storage <-> Trustee Administration Server
  - Storage -> TAS: Delivers election manifest and election data.
  - TAS -> Storage: Returns decrypted tally or updated manifest.

Within the Air-Gapped Network

- Election Administrator <-> Trustee Administration Server
  - Admin -> TAS: Starts key-generation, ballot-decryption, or homomorphically ceremonies.
- Trustee Administration Server <-> Trustee Applications (x N)
  - TAS -> TA: Sends election info, decryption requests; signals threshold reached.
  - TA -> TAS: Signs in; confirms key load; submits shuffled ballots or decryption info.
- Trustee Application <-> Trustee Storage
  - TA -> Storage: Writes trustee key share.
  - Storage -> TA: Loads trustee key share.
- Trustee Administration Server -> Ballot Printer (optional, not pictured)
  - TAS -> Printer: Sends printable ballots for paper tally.

## Architecture as a verifiable blueprint

For our cryptographic voting protocol we chose to create a formal architecture model using the Systems Modeling language (SysML) that specifies exactly how components interact to fulfill requirements. The formality of our architecture model enables automated analysis before implementation begins. We can analyze key properties such as:

- **Information flow security:** Ensuring sensitive data like private keys and plaintext ballots only flow through appropriate channels
- **Protocol correctness:** Validating that interaction sequences (like ballot casting and checking) fulfill requirements
- **Component connectivity:** Confirming that all necessary connections exist without extraneous paths

When issues are found, we can refine the architecture and the rest of the artifacts so far – a far less costly process than discovering flaws later.

**KEY IDEA:** *A verified blueprint prevents structural flaws long before code is written.*

These architectural models form a link in the RDE refinement chain; the architecture must demonstrably refine the domain concepts and requirements, and it, in turn, provides the precise specification that will be refined by detailed design and implementation. The ultimate goal is to achieve *architectural conformance*, ensuring the final implemented system precisely matches this verified blueprint, preventing structural deviations that could introduce subtle but critical flaws and vulnerabilities.

Architecture is therefore a verifiable structural foundation for the system, ensuring the system is not only built right according to specification but also that its structure inherently supports the critical security and trustworthiness requirements defined from the outset.

## Design: From Blueprint to Specification

Having established the system’s architectural blueprint, rigorous digital engineering now tackles a critical transition: turning that high-level structure into a precise specification that developers can implement reliably. This is where *design* enters the RDE process. Design in RDE is fundamentally about refinement: taking high-level architectural concepts and giving them precise, unambiguous meaning.

The goal of design is to turn architectural abstractions into explicit, traceable specifications that leave little room for misinterpretation during implementation, ensuring that the final implementation faithfully realizes the original architectural vision. In traditional software development, design often happens implicitly during coding, with developers making structural and behavioral decisions on the fly. In RDE every significant design decision is explicitly documented and justified.

### Design by Contract: Specifications as Guarantees

Part of RDE’s design approach is Design by Contract (DbC), a methodology that treats every interaction between different components of a system as a formal agreement. There is a parallel to legal contracts here: Just as legal contracts specify the obligations and expectations between parties, software contracts specify the preconditions a component can assume and the postconditions it guarantees to deliver.

These contracts serve multiple crucial purposes:

1. They eliminate ambiguity about component responsibilities.
2. They enable automatic verification with tools that can mathematically prove that

implementations satisfy their contracts.

3. They provide “free” test oracles: the contracts themselves become comprehensive test cases that verify correct behavior far more thoroughly than manually written tests.

The contracts are commonly written in specialized Behavior Interface Specification Languages (BISLs) that are precise enough for mathematical verification, yet readable enough for human review and understanding.

## Verification-Centric Design Philosophy

**KEY IDEA:** *In RDE, correctness outweighs convenience.*

Perhaps the most distinctive aspect of RDE design is its verification-centric philosophy: designing systems specifically to be easy to verify rather than just easy to implement. This illustrates a core RDE tenet: **for critical systems, the ability to prove correctness is more valuable than implementation convenience.**

Verification-centric design influences every design decision. The interaction points between components are kept minimal and precise to reduce the verification burden. Complex behaviors are decomposed into simpler, more verifiable pieces. Dependencies among components are carefully managed to enable compositional verification, proving properties of the whole system by proving properties of its parts.

## Systematic Design Techniques

RDE employs a methodical approach to tackle complex system design by breaking big problems into smaller, manageable pieces. This systematic methodology relies on four key strategies working in concert:

- **decomposing complex systems into manageable components** that can be traced back to their original requirements,
- **cataloguing proven design solutions for reuse** (such as secure data handling patterns in cryptographic systems),
- **building designs incrementally** with verification at each step, and
- **prioritizing correctness over performance** during initial development.

## From Architecture to Implementation

The output of RDE design is a precise “design specification”, or implementation plan. This plan includes:

- **Detailed component specifications** with formal contracts defining behavior
- **Data structure definitions** with invariants that must be maintained
- **Interface specifications** defining exactly how components interact
- **Verification conditions** that implementations must satisfy
- **Traceability links** connecting each design element back to requirements and architectural decisions

This plan provides implementers with unambiguous guidance while enabling automatic verification

that implementations conform to specifications. The design itself becomes a formal artifact that can be analyzed, verified, and evolved systematically.

## Ready for development

By treating design as explicit decision-making with formal specifications and contracts, RDE ensures that the transition from architecture to implementation preserves critical system properties. Design provides the precise implementation plan needed to realize complex, reliable systems like the secure voting protocol.

## Development: Specification-driven Coding

With a precise implementation plan in hand, RDE reaches the *development* phase, where models and specifications finally become executable code. Development in RDE is a disciplined process that maintains the same rigor established in earlier phases. Rather than viewing source code as a separate and isolated artifact, RDE views it as another model in the refinement chain, one that must conform to the same standards of precision and verifiability established in earlier phases.

**KEY IDEA:** *Source code is just another model; it must meet the same rigorous standards as the preceding artifacts.*

## Building from the foundation up

RDE development follows a systematic “structure-first” approach that prioritizes getting the fundamental building blocks right before adding complex behavior.

The process follows this progression:

- **Foundational Elements First:** We implement the fundamental data structures that represent core concepts like encrypted ballots, how they are to be interacted with programmatically, and what rules and constraints they must follow (these are called “types and interfaces”).
- **Built-in Rule Enforcement:** We build the basic operations that create and validate these structures, along with safeguards that ensure the system can never accidentally create invalid data (constructors and data validation).
- **Bottom Up Implementation:** We work our way up from simple, well-understood components and gradually build up to more complex behaviors. Initially, some parts may be placeholder versions that simply indicate “this feature isn’t ready yet,” which we then systematically replace with working versions.

This approach prevents the structural errors that often plague traditional development, where data representations evolve organically during coding and can end up inconsistent with the original design intent.

## Comprehensive Quality Assurance

RDE development includes multiple layers of quality control that provide continuous feedback throughout the coding process:

- **Real-time Monitoring:** We continuously monitor the code as it's being written, checking that it follows established patterns and standards and immediately flagging potential issues and logical errors (this uses “static analysis tools” and “runtime checking”).
- **Automated Code Creation:** Much of the routine, error-prone work is handled by having computers automatically generate code from our precise specifications rather than writing it by hand. This isn't the fuzzy, probabilistic code generation you might associate with AI tools—this is a time-tested, more deterministic approach known as correct-by-construction, where code is derived directly and reliably from formal models, ensuring it does exactly what it's supposed to do.

**Continuous Validation:** Our formal specifications automatically become comprehensive test suites that check thousands of scenarios we might never think to test manually.

This multi-layered approach catches problems immediately rather than hoping they'll surface later during testing.

## Security Engineering: Withstanding Determined Adversaries

Throughout the entire rigorous digital engineering process we have been implicitly pursuing three fundamental pillars: correctness, usability, and security. Let's turn our attention specifically to security. How do we ensure that our carefully engineered systems can withstand deliberate attacks from adversaries who want them to fail? *Security engineering* formalizes a way to make security a “first-class concern” that is part of every aspect of system development from day zero.

### Start with Transparency

One of the most counterintuitive principles in security engineering is that genuine security cannot depend on keeping the internals of a system secret. This principle, rooted in 19th-century cryptographer Auguste Kerckhoff's work, states that “a system should be secure even if everything about the system, except its secrets, is public knowledge.” For the voting protocol, this means that we want to publicly document our entire cryptographic protocol, source code, verification methods, and overall approach, while maintaining the secrecy of specific data like cryptographic keys and information that could be used to violate voter privacy and secrecy.

**KEY IDEA:** *Genuine security cannot depend on the secrecy of a system's implementation.*

This demanding definition explains why so many systems that appear secure on the surface fail when subjected to determined attack. Transparency allows security experts to examine and improve our work, builds public trust, and forces us to build systems that are genuinely secure. This is in stark contrast to what many organizations instinctively want to do: hide their system's inner workings in the hope that obscurity will provide protection. The problem with this “security-through-obscurity” approach is that it requires perfect, permanent secrecy, which is impossible to maintain over time. It takes only a single, dedicated and talented adversary to defeat a system's security forever, once those hidden details are discovered.

### Know Your Enemy: Comprehensive Threat Modeling

A crucial aspect of security engineering is developing what practitioners call *adversarial thinking*, the ability to step into the mindset of someone trying to break your system. This requires systematically

modeling who might attack your system, what they want to achieve, and what capabilities they possess.

In threat modeling, we create what are essentially character profiles for our adversaries. Each archetype has different motivations, resources, and attack capabilities. For a voting system, adversaries might include individual hackers seeking notoriety, organized crime groups pursuing financial gain, and foreign nation-states attempting to undermine democratic processes. The methodology recognizes that complex systems may face hundreds of different potential adversary types, but recommends focusing on key archetypes that represent the extremes of the threat landscape.

Our threat model recognizes several key adversary classes:

- **Compromised devices:** From malware infections to supply chain attacks on voting devices, trustee applications, or backend servers
- **Network adversaries:** Capable of tampering with, adding, or removing protocol messages between system components
- **Corrupt insiders:** Including election administrators, trustees, or cloud providers who might manipulate the system from within
- **Cryptographic attackers:** Who might exploit weaknesses in algorithms, implementations, or parameter choices

For critical systems like voting protocols, RDE adopts what might seem like a paranoid assumption set borrowed from Department of Defense practices:

- The adversary is a nation-state with unlimited resources and talent
- The adversary knows everything about your system's design and implementation
- The adversary is already inside your deployed system with elevated privileges

These assumptions might seem extreme for civilian systems, but we contend that weakening assumptions for critical systems rarely decreases project risk or saves significant time or money.

### **Security Assurance Cases: Attack Scenarios and Mitigations**

A distinctive aspect of security engineering is its insistence on *security assurance cases*. Security assurance cases demonstrate that, given an explicit set of assumptions a set of security properties are guaranteed. For the voting protocol, our assurance case includes:

- A complete hierarchy of 47 specific security requirements organized under confidentiality, integrity, and availability
- A formal threat model explaining 100+ specific adversary capabilities and attack scenarios
- Identification of mission-critical functions and their security dependencies
- Rational, cryptographically-grounded mitigations for each identified threat
- Mathematical security proofs demonstrating that our cryptographic constructions deliver claimed properties

A few examples of attacks and mitigations:

- **Mismatched Encryption Attacks:** A compromised voting device might encrypt a ballot that

doesn't correspond to the voter's actual choices. Our primary mitigation is *cast-as-intended verifiability*: we implement cryptographic mechanisms that allow voters to check their ballots using a separate device to verify the encryption matches their intended choices. For large-scale attacks, statistical detection becomes possible if enough voters perform verification.

- **Ballot Tampering:** Network adversaries might add, alter, or remove encrypted ballots as they travel between system components. We mitigate this through *recorded-as-cast verifiability* (voters can verify their ballots were recorded correctly) and message signatures that provide cryptographic authenticity for all protocol communications.
- **Bad Mixing or Decryption:** Corrupt trustees or compromised trustee applications might incorrectly shuffle or decrypt ballots. Our defense is *counted-as-recorded verifiability* with mathematical proofs of correct shuffling and decryption that can be independently verified by anyone.
- **Side-Channel Attacks:** Adversaries might extract secrets by observing timing patterns, power consumption, or electromagnetic emissions from cryptographic computations. We address this through controlled physical environments for sensitive operations and careful implementation practices that minimize information leakage.

Each attack in our model is explicitly linked to the security properties it threatens and the specific mitigations that defend against it.

## Cryptography: The Mathematical Foundation

Cryptography provides security's mathematical foundation, "the science of keeping secrets" through algorithms that remain secure even when their operation is fully public. A well-known dictatum in the world of cryptography is that developers should never create their own cryptographic algorithms from scratch, due to the complexity and likelihood of something going wrong. Instead, it is recommended to use well-established, peer-reviewed standards that have withstood years of analysis by the global cryptographic community.

For our voting protocol, this meant building upon proven cryptographic components. We deliberately chose algorithms from established suites like the U.S. Government's Suite B and emerging post-quantum algorithms designed to remain secure against future quantum computers. Every cryptographic algorithm we choose is backed by formal analysis and extensive peer review.

The protocol employs sophisticated cryptographic techniques:

- **Threshold cryptography** distributes trust among multiple trustees, ensuring no single party or group below a certain size can compromise ballot secrecy and other important properties.
- **Mix networks** break the link between voters and their encrypted ballots through mathematically verifiable shuffling
- **Zero-knowledge proofs** allow verification of correct protocol execution without revealing sensitive information

## Trust Distribution and Zero Trust Architecture

**KEY IDEA:** *Every component must prove itself; zero trust is the rule.*

Modern security recognizes that placing trust in any single component is a vulnerability. Our voting protocol employs *trust distribution* as a core security strategy. We distribute critical operations across multiple trustees using threshold cryptography. Even if some trustees are compromised or corrupt, the protocol remains secure as long as a sufficient threshold of honest trustees participate.

This aligns with Zero Trust principles – the recognition that a system’s components must not trust each other by default. Every component must prove its identity and authorization for each interaction through cryptographic mutual authentication and access control. For example, in our protocol, trustees must cryptographically sign all protocol messages, making it impossible for corrupt election administrators to forge trustee actions without compromising their private keys.

## Security Throughout RDE

Security engineering permeates every other aspect of RDE. We make security concepts a part of the fundamental vocabulary in Domain Engineering; terms like “ballot secrecy,” “cast ballot,” and “adversary” receive precise definitions that eliminate ambiguity. We capture security requirements with the same rigor as functional requirements, tracing each back to specific threat scenarios. We incorporate security properties into the structural design through formal architecture models that can be analyzed for information flow and access control properties. And in product line engineering, we extend this security analysis across families of system variants, identifying which security properties hold universally across all configurations and which feature combinations might introduce vulnerabilities requiring additional mitigations.

This integration addresses a fundamental challenge in traditional security approaches: the disconnect between security and development teams. In RDE, security isn’t something other experts “do to” a system. It’s a fundamental design consideration that shapes every engineering decision.

## Verification and Validation: Proving Systems Work

After careful domain modeling, precise requirements engineering, and rigorous development, we arrive at a crucial question: How do we know our carefully crafted system actually works as intended? This is where *verification and validation* (V&V) enters the RDE process.

Like security engineering, RDE treats V&V as an integral part of every phase. Rather than asking “Does our finished system work?” RDE continuously asks two more precise questions throughout development:

- **Are we building the right system?** *Validation* asks whether we’re solving the correct problem: does our voting protocol actually meet election officials’ needs and voters’ expectations? Does it address the real security threats we identified? Will election technology vendors be able to use it successfully in practice?
- **Are we building the system correctly?** *Verification*, on the other hand, asks whether our solution is mathematically and logically sound. Given our requirements and design specifications, does our implementation actually fulfill them? Is our cryptographic implementation correct? Do our security mechanisms genuinely provide the protection they claim?

## Models as a Foundation for Testing

**KEY IDEA:** *RDE keeps asking: are we building the right system, and are we building it right?*

RDE leverages the formal models created throughout development as the foundation for comprehensive testing. Rather than manually writing thousands of individual test cases, our models systematically generate test cases that cover all the behavioral patterns we specified, and we don't have to hope we remembered to test every important scenario.

This approach reveals a deeper technical connection that's worth understanding because it illuminates why RDE is so powerful: in formal systems, mathematical theorems can be directly transformed into executable property-based tests. This reflects a fundamental relationship known as the Curry-Howard correspondence—the idea that mathematical proofs and computer programs are actually two sides of the same coin. When we prove a theorem about our system's behavior, we're simultaneously creating a program that can test whether that behavior holds in practice.

This connection enables sophisticated coverage analysis: we can measure not just whether our tests exercise every line of code (traditional coverage), but whether they explore every behavioral pattern in our formal models. We can quantify gaps between what we've verified mathematically and what we've tested empirically, giving us precise insight into where our assurance is strong versus where additional validation work is needed.

This mathematical foundation transforms testing from guesswork into rigorous science. It proves especially valuable for cryptographic systems, where subtle errors and corner cases can completely undermine security.

### Comprehensive Verification Strategies

RDE employs multiple verification techniques of increasing mathematical rigor, each providing different types of confidence in system correctness.

- **Static Analysis:** Specialized tools analyze our code without executing it, automatically detecting potential security vulnerabilities, memory management errors, and violations of coding standards.
- **Model Checking** takes verification a step further by systematically exploring system behaviors through mathematical analysis. Rather than testing specific scenarios, model checkers examine every possible sequence of events our system might encounter within specified limits. For finite systems, this can provide strong evidence that certain system properties hold no matter the circumstance.
- **Formal Verification** represents the highest level of rigor: mathematical proof that implementations satisfy their specifications. Using theorem provers, automated software tools that can verify mathematical proofs, we can verify that our cryptographic protocols provide the security properties we claim, even against adversaries with unlimited computational resources.

### Validation Throughout Development

RDE integrates validation activities directly into the developer workflow through continuous

automation. All verification and validation techniques run automatically on developer workstations and within “continuous integration” pipelines, providing immediate feedback as code evolves. This creates a “tied validation” approach where code structure, documentation, specifications, and tests must all be consistent and realizable before expensive formal verification techniques are even attempted, preventing wasted effort on fundamentally flawed implementations.

The development process deliberately begins with “bottom” implementations: minimal versions that may crash or fail to terminate, specifically designed to validate that specifications are sound and implementable. As developers systematically refine these implementations toward full functionality, the continuous validation framework provides precise guidance on development priorities: which components are complete, which require additional work, and which parts of the system currently have strong versus weak assurance.

### **Building Genuine Confidence**

The comprehensive V&V approach in RDE creates something rare in software engineering: genuine confidence in system behavior. Rather than hoping our extensive testing caught the important problems, we have strong evidence that critical properties hold. Rather than assuming our implementation matches our intentions, we have verified traceability from high-level goals down to specific code..

For critical systems like voting protocols, this level of assurance isn’t academic luxury—it’s essential for public trust. A voting system that provides mathematical proofs of ballot secrecy and cryptographic integrity, which security experts can independently verify, has the credibility necessary for democratic processes

RDE moves critical systems development from a craft based on experience and intuition to a science based on mathematical certainty. The result is systems we can genuinely trust, not because we hope they work, but because we can prove they do.

### **Evolution and Maintenance: Preserving Trust Over Time**

Mathematical proofs and rigorous verification provide tremendous confidence in a system’s initial correctness, but what happens when that carefully engineered system must change? How do we maintain trustworthiness over years or decades of deployment?

Systems developed with RDE present both unique challenges and powerful advantages when it comes to evolving and maintaining systems. The same rigorous artifacts that enable mathematical assurance (formal models, precise specifications, verified proofs) must themselves be maintained as the system evolves. This creates complexity, but also provides unprecedented visibility into how changes ripple through a system.

### **The Inevitability of Change**

Real-world systems face relentless pressure to evolve. For a voting protocol that will be the core building block of future voting systems, these pressures might include new accessibility requirements mandated by legislation, security enhancements responding to emerging threats, or the need to support different ballot formats across jurisdictions. Platforms we use in the development of the protocol, such as cryptographic building blocks, might also receive updates and require the protocol implementation to change.

Each category of change presents challenges. Feature enhancements may require extending our domain model with new concepts and updating models to account for new attack surfaces.

Changes to dependencies can force updates to specifications when the new version of the dependency provides different guarantees than the old one. Even seemingly simple bug fixes can cascade through formal specifications and verification artifacts in ways that traditional development rarely experiences.

### The RDE Advantage

The very characteristics that make RDE systems more complex to maintain also provide powerful tools for managing that complexity. Compositionality (the principle of building systems from well-defined, independent components) means that changes often remain localized rather than propagating unpredictably throughout the system.

**KEY IDEA:** *Systems change; traceability preserves trust over decades.*

More importantly, the extensive traceability built into RDE artifacts provides critical information that traditional systems lack: precise understanding of change impact. When a cryptographic library updates the way it is meant to be interacted with, automated tools can identify exactly which specifications, proofs, and implementations require review.

### The Long-Term Challenge: System Ownership

Perhaps the most overlooked aspect of critical systems is ownership change. Voting systems, like all infrastructure, outlive their original creators. Institutional knowledge disperses with team changes, company acquisitions, etc. Traditional systems often become unmaintainable and new team members are forced into expensive rewrites or dangerous continued operation of poorly understood code. RDE's comprehensive documentation provides unusual resilience against this knowledge loss. The traceable models create a complete record of design decisions and their justifications.

The mathematics underlying our voting protocol will remain valid for decades, but the systems implementing that mathematics will inevitably change. RDE allows for preserving trustworthiness through that evolution and throughout the systems' operational lifetime.

## Applying Rigorous Engineering: The Promise and the Challenge

The advantages of RDE extend beyond any single project. In our experience, teams that adopt these methods deliver outcomes that seem almost too good to be true: dramatically fewer bugs discovered during testing and shortened development timelines.

**KEY IDEA:** *RDE catches problems when they cost pennies instead of millions.*

Similarly, in our experience the rigorous upfront investment in modeling and specification typically *reduces* total project costs. RDE shifts problem-solving “left” in the development timeline, when solutions are cheapest and most effective. Problems caught during domain modeling cost pennies to fix; the same issues discovered after deployment can cost millions.

The precision of RDE artifacts also completely changes how teams communicate. When everyone uses the same carefully defined vocabulary, misunderstandings are dramatically rarer.

## The Adoption Paradox

If approaches like RDE deliver such clear advantages, why does they remain relatively uncommon? The challenge is rarely technical: the tools and methods are mature and proven. Instead, the barriers are fundamentally human and organizational.

- **Cultural resistance:** Engineering culture naturally resists change. Teams comfortable with familiar processes often view RDE's systematic approach as unnecessary overhead, especially when their current methods "work well enough." This resistance deepens when RDE initially appears to slow development, even though it typically accelerates delivery overall. The most successful RDE adoptions recognize that transformation requires social and cultural solutions alongside technical ones. Change typically succeeds when respected team members become convinced advocates who demonstrate concrete wins. Early projects that show clear value build confidence and momentum for broader adoption. Leadership support is essential, but mandates alone rarely succeed. Sustainable change emerges when teams experience the benefits firsthand: catching subtle bugs early, clarity during development, and systems that work correctly the first time.
- **Skillset:** The methodology also requires skills that require some upfront investment. The approaches are designed to hide the mathematical complexity behind user-friendly approaches while preserving the underlying rigor, but some familiarity with formal modeling, specification languages, and mathematical verification makes applying RDE much more effective. The average developer is not exposed to those skills. Organizations must accept investment for teams to climb the learning curve.
- **Tooling:** The current state of RDE tooling represents perhaps the most significant practical barrier to RDE adoption. Publicly available tools require extensive expertise and steep learning curves that make them inaccessible to most developers. Even experienced practitioners find themselves implementing substantial functionality by hand, as existing tools cover only a fraction of what's needed for practical RDE workflows. Integration with standard development environments remains challenging and often requires custom solutions. These practical friction points can discourage adoption, particularly in organizations focused on short-term delivery pressure.

DARPA—the organization that has historically supported breakthrough advances that transform everyday life, from the internet to GPS—has recognized these integration challenges as a critical barrier to national security. They are now investing in tools and corporate infrastructure to support the adoption of formal methods and other approaches we have discussed here across the U.S. Government and beyond. Their Resilient Software Systems Accelerator provides seed funding to help defense industrial base companies integrate rigorous engineering tools, directly addressing the practical challenges of enterprise adoption.

## The Emerging Mainstream

Despite adoption challenges, the mindset and approaches underpinning RDE are gaining momentum, driven by converging forces. Government initiatives like the Department of Defense's Digital Engineering strategy explicitly promote model-based approaches to reduce costs and improve outcomes. Most recently, DARPA has made resilient software systems a strategic priority. At DARPA's 2025 Resilient Software Systems Colloquium, Deputy Director Rob McHenry declared the initiative "most likely to be DARPA's impact of the decade," emphasizing that "it simply is no

longer OK to accept the risk of cyber-vulnerabilities.” Resilience *is* RDE in this context, and we are encouraged to see the support of the mathematical rigor and systematic approaches we have outlined here.

**KEY IDEA:** *Rigorous methods are shifting from research labs to boardrooms.*

We also believe competitive pressures are making RDE adoption inevitable for critical systems. Organizations that can deliver reliable systems faster and cheaper than competitors will win contracts and markets. Software systems have become central to critical infrastructure, and developers can no longer afford superficial security claims. Market forces and real-world consequences quickly expose any gaps between promised and actual reliability. The cost of rigorous development is typically far less than the cost of failure. As these methods become more accessible and widely understood, the advantages will become too significant to ignore.

## Voting Protocol: What Lies Ahead

Our cryptographic voting protocol provides a mathematically verified foundation upon which others can build secure voting systems. While the protocol’s core cryptographic operations have been rigorously engineered and proven correct, the voting systems that will be built on top remain vulnerable to implementation errors, user interface flaws, and integration mistakes.

**KEY IDEA:** *A provably secure core is only the beginning; vigilance must continue in every implementation.*

Vigilance must continue as vendors and jurisdictions develop the full systems that voters will actually use. We hope this work demonstrates that rigorous digital engineering can deliver the trustworthy systems our democracy demands, inspiring others to apply these same methodologies across the broader ecosystem of critical infrastructure we all depend upon.

## The Stakes Are Rising

The consequences of software failure in critical systems are not measured in inconvenience and money, but in human lives and societal stability. This reality demands a fundamental break from the “move fast and break things” mentality that has shaped much of the software industry.

This urgency is amplified by a troubling asymmetry: sophisticated adversaries now use formal methods and systematic analysis to find vulnerabilities in critical systems, while many of those same systems were built using traditional development approaches. When attackers employ rigorous mathematical tools to exploit systems developed without such rigor, the defender is fundamentally outmatched. For systems of national significance like voting protocols, this asymmetry makes the transition to rigorous methods essential.

We believe rigorous digital engineering offers a path forward: treating critical systems development as the engineering discipline it must become. The question is not whether critical systems engineering will become more rigorous—it’s whether we will make that transition quickly enough to address the challenges ahead. For systems of national significance like our voting protocol, that transformation cannot come soon enough.

*The voting protocol and all development artifacts are open source and available on Github, [here](#).*