Contents

The Colorado RLA Tool Book

Free & Fair

Phase 2 Release—August 2017

License

This project will be licensed under an OSI-approved license. Discussions are ongoing with the Colorado Department of State about which Open Source license is most appropriate for this system.

- TBD: justification for license choice

Installation and Use

Downloading

The Colorado Risk-Limiting Audit (CORLA, for short) system is open source and freely available via GitHub. One can download the entire project, including all development artifacts and source code, via the GitHub Colorado RLA project webpage, either by cloning its git repository or as an archive file.

We also automatically build a distribution archive using the Node package manager (for the front-end) and an Maven-based Java archive file compilation (for the back-end). See the client/README.md for more information about the former, particularly the `npm run pack` command. See the [server/README-ECLIPSE.md] file for more information about the latter.

Installation

This system is installed at the moment by installing a Postgres database and running the server in a Java 8 Virtual Machine (JVM).

Database installation and setup, which is a one-time operation, is documented in the *Data Persistence* section of the *Developer Instructions* chapter of this book.

Example of Use

See the User Manual and Run Book.

Running the latest development version

For testing purposes, you can run the latest versions of the client and server directly from a clone of the git repository. One must configure the database as discussed previously and run both the client and the server to test the CORLA system.

For the server, follow the directions in `server/README-ECLIPSE` to install Eclipse, and use Eclipse's Run menu.

Building and packaging the server is accomplished by running `mvn package` from the `../server/eclipse-project` directory, as in

```
cd server/eclipse-project
mvn package
```

Running the server can also be accomplished by running the application directly (as in `java -jar ../server/eclipse-project/target/colorado_rla-VERSION-shaded.jar`,

for some specific *VERSION* number) or installing the system in a webserver, as documented in the `INSTALL.html` documentation delivered with the system to CDOS.

To run the client in a standalone manner (rather than from a deployed server), you will need to get Node.js and npm. Next, run:

```
cd client
npm install
npm start
```

Then visit `http://localhost:3000/` in a supported web browser and you will be able to authenticate to the CORLA system.

Developer Instructions

This document is for developers interested in contributing to this project. It describes the technologies used, the modules and libraries on which we depend, and how to build the system. It also covers how to perform quality assurance, validation, verification, and deployment of the system.

Determining if the system is correct includes both checking behavioral properties and non-behavioral properties. Behavioral properties are all about correctness and boil down to decidable (*yes or no*) questions about the system. Non-behavioral properties are measurable—such as how many resources the system uses, how reliable it is, or whether it is secure—and checking them entails ensuring that measures are as we specify. More specifically, we cover system performance and reliability.

Finally, the document closes with a link to our project dashboard.

History

- Outline and first draft, 5 July 2017 by Joe Kiniry.
- Second draft that is mostly textually complete, 6 July 2017 by Joe Kiniry.
- Third draft with updates for phase-1 delivery, 17 August 2017 by Joe Kiniry.
- Fourth draft with updates for phase-2 delivery, 24 August 2017 by Joe Kiniry.

Platform and Programming Languages

To fulfill Colorado's requirements, we use a modular system design and standard, well-understood web application technologies.

We are using a Java-based web application running on a Linux server, hosted in the CDOS data center. Deployment is targeted for a JVM version that supports Java 8.

The choice of deployment on JVM is made due to IT constraints on the part of CDOS. (See page 11, "Hosting Environment", of the DQ.) While we could have developed on .Net, and there are very good tools for rigorous engineering on such (which we have used in, e.g., our electronic poll book demonstrator), there are still significant challenges in multi-platform development and deployment. We would rather have a straightforward cross-platform system development and deployment story, thus we use Java 8 on the server-side.

We are using PostgreSQL, via the Hibernate ORM, for data persistence.

The user interface (UI) is browser-based. The client is written in TypeScript, a mainstream, Microsoft-supported variant of JavaScript that offers opt-in, Java-like type safety. TypeScript compiles to plain, human-readable JavaScript, so this choice supports client-side correctness without requiring any special web browser support.

Developer Tools

We are using many of the following tools. Developers need not download and install this long list of technologies. We have provided developers with a pre-configured Eclipse install, a bootstrapping Eclipse workspace, and a VM image which can be loaded into VirtualBox or other comparable virtualization platform. We provide these resources in order to both decrease new developers' ramp-up time as well as to standardize on specific versions of tools for development.

Instructions for installing Eclipse and automatically installing and configuring the plugins we use are found in README-ECLIPSE.md.

- GitHub for distributed version control, issue tracking, and development documentation
- the PVS specification and verification system or the Alloy tool for specifying and reasoning about formal domain models and automatically synthesizing system tests
- the Software Analysis Workbench (SAW) for formal verification of intermediate representations and reasoning
- the CheckStyle lightweight static checkers for ensuring code standard conformance,
- the FindBugs lightweight static checker for code quality evaluation,
- the PMD lightweight static checker for code quality evaluation,
- the Java Modeling Language (JML) for formally specifying the behavior of our Java implementation
- the OpenJML tools suite for performing runtime verification, extended static checking, and full functional verification of Java implementations against JML specifications
- the JMLunitNG tool for automatically generating test benches from JML-annotated Java code
- the Coq proof assistant for formally specifying and reasoning about various formal models of the system and elections in general
- (optionally) the KeY tool for performing full functional verification and test case generation of implementations with JML formal specifications
- (optionally) the Cryptol tool for specifying and reasoning about cryptographic algorithms
- (optionally) the F* tool and Tamarin prover for specifying and reasoning about cryptographic protocols
- the Gnu Compiler Collection (gcc), the clang compiler, and (optionally) the CompCert compiler for compiling C code

- various automated theorem provers such as Z3, the Lean theorem prover, CVC4, Yices, and ABC for automatically reasoning about formal models
  * Emacs, Eclipse, IntelliJ IDEA, and other JetBrains technologies for Integrated Development Environments,
- (optionally) the Community Z Tools (CZT) supporting the Z formal method, the Rodin platform supporting the Event-B formal method, the Overture tool supporting the VDM formal method, and the RAISE tool supporting the RAISE specification language (RSL) for specifying and reasoning about formal models of systems
- the OpenJDK Java developers kit
- JMLUnitNG for automatic test code generation
- standard test coverage tools such as JCov and Cobertura
- (optionally) Java PathFinder (JPF) and similar model checkers for reasoning about safety properties of implementations
- OmniGraffle for drawing diagrams
- various BON-related tools, including the BONc and FAFESSL tool suites, which are based upon the BON method, for system specification
- the Beetlz tool for refinement checking of BON specifications against JML-annotated Java implementations
- (optionally) ProVerif, UPPAAL, and the TLA+ tools for distributed algorithm specification and reasoning
- the TypeScript language for front-end development, using the React UI framework
- *TBD Daikon*
- *TBD AutoGrader*
- and Travis CI for continuous integration

Dependencies

TBD: A concrete list of dependencies, preferably at the module/library level, complete with versioning information. Note that we prefer that this dependency list is automatically generated and kept up-to-date by the build system.

Spark

TBD: Describe Spark and its use.

Data Persistence

In order to use the Postgres database in development, one must:

1. Install PostgreSQL (`brew install postgres` on MacOS, `apt-get install postgresql` on many Linux distributions, or whatever is appropriate) and start it running.

2. Create a database called "`corla`", and grant all privileges on it to a user called "`corla`" with password "`corla`".

3. Initialize the "`corla`" database with test administrator data. For example, to accomplish the above on MacOS using Homebrew, one issues the following commands:

   ```
   brew install postgres
   createuser -P corla
   createdb -O corla corla
   ```

   On Linux, one would replace the first command with something akin to `sudo apt-get install postgresql`.

That's it. If the database is there the server will use it and will, at this stage, create all its tables and such automatically. 4. Run the server (to create all the database tables). Recall that this is accomplished by either running the server in Eclipse using the Run button or running it from a command line using a command akin to `java -jar colorado_rla-VERSION-shaded.jar`. 5. Load test authentication credentials into the database, by executing the SQL in `corla-test-credentials.psql` (found in the `test` directory of the repository). This can be done with the following command on OS X:

```
psql -U corla -d corla -a -f corla-test-credentials.psql
```

or the following command on Linux:

```
psql -U corla -h localhost -d corla -a -f corla-test-credentials.psql
```

If you need to delete the database—perhaps because due to a recent merge the DB schema has evolved—use the `dropdb corla` command and then recreate the DB following the steps above.

There are helpful scripts for automating these actions located in the `server/eclipse_project/script` directory.

TBD: Describe the Hibernate ORM and its use.

Communication Formats

TBD: Describe the Google GSon library and tools.

Code Review and Source Management

We use the Git SCM for version control, with GitHub for hosting and code review. The development worfklow is as follows:

1. Locally, pull the latest changes on the `master` branch of the upstream repo, hosted on GitHub.

2. Check out a new topic branch based on the above changes.
3. Commit changes to your local branch.
4. For the sake of visibility, you may open a work-in-progress Pull Request from your branch to `master`. If you do, add the `wip` label.
5. When you are ready to merge to `master`, make sure your branch has been pushed to the remote repository and open a Pull Request (if you haven't already). Remove any `wip` label and add the `review` label.
6. If appropriate, use the GitHub "Reviewers" dropdown to formally request a review from a specific person. Either way, paste a link to the PR in Slack to alert others who may wish to review it.
7. At least one other person must review any changes to the `master` branch and approve it via the GitHub PR interface comments. A *reviewer* should check that all new commits are signed, and all necessary comments are addressed.
8. Before it can be merged, you will generally have to `rebase` your branch on to the `master` branch in order to preserve a clean commit history. You can do this with commands in your branch: `git fetch`, then `git rebase origin/master` (addressing any merge conflicts if necessary), and finally `git push --force-with-lease origin <yourbranch>`.
9. Note that *force-pushes can be dangerous*, so make sure that you know that no one else has pushed changes to the branch which aren't in the history of your branch. If others on the team are pulling and testing it locally, they will need fix up their local branches with `git checkout <yourbranch>`, `git fetch`, and `git reset --hard origin/<yourbranch>`. For more details, see The Dark Side of the Force Push - Will Anderson and –force considered harmful; understanding git's –force-with-lease - Atlassian Developers
10. Finally, a *reviewer* with merge permissions can merge the PR using the GitHub "Merge pull request" button. This will introduce an *unsigned* merge commit, but preserve the signatures on the actual branch's commits. Finally, the PR submitter, not the reviewer, should delete the merged branch.

Guidelines:

- Do not commit directly to `master`.
- To support bisecting, do not merge WIP commits that break the build. On topic branches, squash commits as needed before merging.
- Write short, useful commit messages with a consistent style. Follow these seven rules, with the amendment that on this project, we have adopted the convention of ending the subject line with a period.
- Keep your topic branches small to facilitate review.
- Before merging someone else's PR, make sure other reviewers' comments are resolved, and that the PR author considers the PR ready to merge.
- For security-sensitive code, ensure your changes have received an in-depth review, preferably from multiple reviewers.
- Configure Git so that your commits are signed.
- Whenever possible, use automation to avoid committing errors or noise

(e.g. extraneous whitespace). Use linters, automatic code formatters, test runners, and other static analysis tools. Configure your editor to use them, and when feasible, integrate them into the upstream continuous integration checks.

- WARNING: sub-projects (e.g. the client, server) should *not* directly depend on files outside of their directory tree. Our CI is configured to run checks only for projects that have had some file changed. If you must depend on out-of-tree files, update `.travis.yml` to avoid false positives.

Building

TBD discussion of which build systems we use and why.

We provide both an integrated Eclipse-based build system and a traditional Make-based build system. The former permits us to support a rich and interactive design, development, validation, and verification experience in an IDE. The latter facilitates cross-platform, IDE-independent builds and continuous integration with Travis CI.

The Eclipse-based build system is built into our Eclipse IDE image and our workspace, as specified in `server/eclipse.setup`.

The Make-based system is rooted in our top-level Makefile. That build system not only compiles the RLA tool, but also generates documentation, analyzes the system for quality and correctness, and more. *(Ed. note: The make-based build system has not yet been written.)*

See the instructions in the **Installation and Use** chapter on running the development system.

Note that the production client build configuration expects server endpoints to have an `/api` path prefix. To support user testing, we currently enable browser console logging in all builds.

Quality Assurance

TBD discussion of the various facets of quality and how these ideas are concretized into metrics and measures that are automatically assessed and reported upon, both within the IDE and during continuous V&V.

We measure quality of systems by using a variety of *dynamic* and *static* techniques.

Dynamic analysis means that we run the system and observe it, measuring various properties of the system and checking to see if those measures are in the range that we expect. We say "range" because many measures have a sweet spot—a good value is not too high and not too low. Running the system means

that we either execute the system in a normal environment (e.g., a Java virtual machine) or we execute a model of the system in a test environment (e.g., an instrumented executable or a debugger).

Static analysis entails examining a system *without* executing it. Static analysis that only examines a system's *syntactic* structure is what we call lightweight static analysis. For example, the source code's style and shape is syntactic. Static analysis that examines a system's *semantic* structure is what we call heavyweight static analysis. For example, theorem proving with extended static checking is heavyweight static analysis.

Each kind of static analysis results in a *measure* of a *property*. Decidable properties are either *true* or *false*, thus a good measure for a property is simply "yes" or "no". Other static analyses have more interesting measures, such as grades ("A" through "F") or a number.

In order to automatically measure the quality of a system, we define the set of properties that we wish to measure and the what the optimal ranges are for the measure of each property. We automate this evaluation, both in the IDE and in continuous integration. The current status of our continuous integration checks is displayed via a dynamic status image on our repository's home page.

Also, we have a tool called the AutoGrader that automatically combines the output of multiple analyses and "grades" the system, and consequently its developers. By consistently seeing automated feedback from a set of tuned static analysis tools, developers quickly learn the development practices of a team and a project and also often learn more about rigorous software engineering in general.

Validation and Verification

TBD high level discussion here about how the goals and technologies discussed in the V&V document are realized.

Determining whether the system you are creating is the system that a client wants is called *validation*. *Testing* is one means by which to perform validation. Mathematically proving that a system performs exactly as specified under an explicitly stated set of assumptions is called *verification*.

Some of the quality assurance tools and techniques discussed above are a part of validation and verification.

The Validation and Verification document focuses on this topic in great detail.

Deployment

Free & Fair develops open source software systems in full public view. Therefore, all artifacts associated with a given project or product are immediately available to all stakeholders, at any time, via a web-based collaborative development environment, such as our GitHub organization. This means that various versions of the same system (e.g., builds for various platforms, experimental branches in which new features are being explored, etc.) are immediately available to anyone who browses the project website and clicks on the right download link, or clones the repository and builds it for themselves.

Delivery of production systems to a client or stakeholder is accomplished by providing the modern equivalent of "golden master disks" of yesteryear. The nature of these deliveries differs according to decisions made during contracting and development, in tandem with the client.

For example, if the deployment platform is a flavor of Linux, one of the standard software packaging systems such as RPM or dpkg is used to deliver products. If the deployment system is Microsoft Windows or Apple OS X, the standard open source packaging software is used to deploy production systems.

System Performance

TBD discussion of automated performance testing

System Reliability

TBD discussion of automated deployment reliability testing

To ensure business continuity, we are applying techniques we have been developing since the 1990s to create systems for clients requiring no more than 0.001% downtime. These techniques include practical applied formal methods (the application of mathematical techniques to the design, development, and assurance of software systems) and a peer-reviewed rigorous systems engineering methodology. Our methodology was recently recommended by a NIST internal report (IR 8151 "Dramatically Reducing Software Vulnerabilities"), presented to the White House Office of Science and Technology at their request in November, 2016.

Project Dashboard

We use Travis CI for continuous integration. It produces new builds for every check-in and runs our test suites, as specified in our `.travis.yml` file and in the `ci` directory.

Current build status for the `master` branch status is reflected in the `README.md` of our main GitHub page. Build status logs, can be found at our dashboard: FreeAndFair/ColoradoRLA.

The raw build logs provide details on which versions of our build tools and libraries were used each step along the way.

Bibliography

TBD add references to appropriate papers

Project Management

The time-to-delivery of this project is extremely short, so it is critical that we take an efficient, economical approach to building the proposed RLA software tool. We view the RLA tool specified in this DQ as a reimplementation and extension of our existing, open source RLA product demonstrator, OpenRLA. The Free & Fair team has extensive pre-existing RLA software development experience and deep domain knowledge. This experience, together with a grounding in lightweight formal methods for high-assurance software engineering, will enable us to build a secure, user-friendly RLA application within the limited timeline.

Project Management Practices

In this section we review Free & Fair's project management practices, which we have used to deliver millions of dollars worth of high assurance systems on time and under budget. Our core project management principles focus on Customer Caretaking, Social Contracts, Continuous Improvement, Artifacts and Evidence, and Transparency.

Customer Caretaking

For all projects we have a dedicated Free & Fair team member whose role is to represent the interests of the client to others at Free & Fair. They are actively engaged with the client and have a role in all project management decisions. They build a deep trust relationship with the client's key performers. This position is a reflection of the trust relationship between us and our clients.

Social Contracts

Our systems engineering artifacts capture technical interdependencies between project team members, but the glue that holds the team together and makes the team work well is our collective social contracts. Our performers explicitly discuss and acknowledge client-supplier relationships between team members and always perform to exceed not only the expectations of our external client (in this case, the Colorado Department of State), but also each internal client (another team member).

Continuous Improvement

Social contracts are renegotiated frequently and fluidly and are directly reflected upon immediately upon completion. For example, at the end of a thirty minute stand-up meeting discussing a milestone that we just reached and what comes

next, we often have a five minute discussion about what worked well and where improvements can be made with regards to that particular piece of work. In particular, we focus on its embedded social contracts. The individuals in our organization always attempt to maximize efficiency, impact, and joy at work.

Artifacts and Evidence

We focus on artifacts and evidence in a project or product. "Meta" aspects like processes and checklists serve meaningful outcomes. This focus on the meaningful is pervasive. Principles trump rules. For example, provable security is mandatory; "security theater" is prohibited.

Transparency

Finally, whether it is with regard to our technology, business practices, or project management approach, transparency is the core principle by which we operate. Telling each other, and the client, when something is working well or working poorly, early and honestly, is common. If necessary, we will tell a client that a technical direction they are excited about is inappropriate and provide objective evidence to justify that conclusion. We always keep the client informed, whether we are ahead of the game or behind the eight ball. In all aspects, and for all projects, we believe that transparency is the keystone of our operation. Without it, our election systems cannot be trustworthy and will not be successful.

Project Management Structure and Responsibilities

Dr. J. Kiniry holds final responsibility for the success of this project. Dr. Zimmerman and Dr. Dodds, working with Dr. J. Kiniry, will write the system specification, design and verify the client/server communication protocol and the server/server synchronization protocol, and implement the server-side and communication subsystems, including the audit computation subsystem and the datastore subsystem. Ms. Miller will work with Mr. Ranweiler and Mr. McBurnett on the UX of the system and will mock up UIs. Mr. Ranweiler is responsible for designing and implementing the client side of the system against the system specification and the UI/UX design. Mr. McBurnett will provide domain expertise in Colorado elections and ballot-level comparison risk-limiting audits, will red team system architecture, design, and implementation, and will perform Q/A on the tool, and will help write documentation. He will also coordinate with the EVN CORLA2 team to get statistical algorithm input, advice, and feedback, and will be on call during deployment for the trial runs of the tool during Logic and Accuracy testing and after the election until the audits are done. Mr. M. Kiniry will write the user guide for the system and will revise

the developer's documentation. Dr. Singer will be the customer caretaker and project lead.

Development Process and Methodology

We detail below our rigorous systems engineering and software engineering process and methodology. We make objective decisions about the appropriate programming languages, tools, and technologies for each project or product. Our toolbox, especially in matters related to rigorous systems engineering and applied formal methods, is broader and deeper than that of any other company in the world.

History

- Outline and first draft, 5 July 2017 by Joe Kiniry.

The Free & Fair Development Methodology

The specific development methodology we use for all of our software is a variant of Design by Contract with some aspects of a Correctness by Construction approach. Our process, method, tools and technologies span several deployment and development platforms, specification and programming languages, and communication and coordination schemes. In short, we use a combination of the following methodologies:

- Correct-By-Construction
- Design-By-Contract
- Refinement-Based Process
- Kiniry-Zimmerman Methodology
- Business Object Methodology
- Formal Hardware/Software Co-Design
- Formal Methods (Alloy, CASL, Event B, RAISE, VDM, Z)

The design specified in the RFP includes security, fault tolerance for robustness, and scalability. Our software development process emphasizes those same qualities. This type of development process has been used to develop hundreds of millions of dollars' worth of military, aviation, biomedical and financial systems that must not fail, because human lives and billions of dollars hang in the balance. We believe that election systems are just as important, because protecting democracy protects human lives and our whole economic system.

Our systems are all fault tolerant and have sufficient redundancy, both in algorithm design and physical architecture, to ensure that they can survive the simultaneous failure of multiple machines or networks.

We will apply the same high-assurance techniques to ensure that this project is developed not only with generic coding best practices, but also with best practices for systems critical to homeland security and medical applications.

The system will be far less prone to failure than even the best standard office software.

One important coding best practice for critical systems is performing a machine-checked functional verification of the core algorithms of the software. We first design a mathematical model that is as easily understood as the English language specification. We then provide an implementation that is mathematically proven to meet the specification. This mathematical proof can be automatically checked on a computer, giving unparalleled assurance that the software is correct. These techniques have historically been used for safety-critical systems, where the failure of a system would result in loss of life (e.g., flight control systems at Airbus) or have enormous cost implications (e.g., failure of a mission to Mars).

By combining these techniques, we create a chain of correctness that starts with the high-level system specification and traces down to the smallest implementation details of the most critical parts of the system. At each step in the chain we focus on providing evidence of correctness, generally in multiple forms, including refinement proofs from informal to formal specifications, unit test suites, and mathematical proofs of correctness and security. In other words, all the effort we put into ensuring that our system is correct generates tangible evidence that gives external parties (e.g. certification labs, security experts, political parties, and the American public) the same confidence in our software that we have.

We strictly adhere to well-documented code standards for the various programming languages in which we develop software, and we use appropriate development and testing environments (IDEs, continuous integration systems, issue trackers) to support our development efforts. We aggressively employ techniques such as linting (automated syntactic checks to catch early programming errors), static analysis, and automated testing to provide continuous feedback on our software development practices.

Our formal domain models provide a high-level view of the logical and modular design of the system, ensuring robustness and scalability. From this view, it is easy to see how the components communicate and what their interdependencies are. It is easy to detect components that are too tightly coupled, which would make future replacement or revision challenging; a loosely-coupled system allows for easy extensibility. Our domain models also make it clear what interfaces need to be satisfied by any future module replacement. This means that there is no danger of swapping out a module for a replacement that does not have all the expected functionality. Once the domain model is satisfactory, we continuously use analysis tools to guarantee that all code we write conforms to the model.

We also create formal models of every data format that we use, both internally and externally. We can use these formal models to generate software that allows a variety of programming languages to communicate natively via our data formats, providing fluent interoperability.

Our designs are always highly modular. Each module uses only open data formats for communication, resulting in a system that can easily integrate with third

party systems and can be modified and upgraded by anyone who is familiar with the data formats. A modular architecture assists with validation and verification, allows for experimentation with user experience variants, and enables phased user acceptance testing. It can also ease customization of the system, allowing new voting methods and audit protocols to be swapped into the system as needed without requiring system-wide changes.

Our development repositories contain the code under development, the full set of development artifacts described above, and unit, performance, and integrated functional test suites for each subsystem and for the system as a whole. Our test servers pull from these repositories and perform automated builds and testing whenever code is updated. In addition to standard functional tests we place a particular emphasis on performance tests, which allow us to ensure that feature changes do not impact performance.

Example Use of Our Methodology

Historically, this kind of system has been evaluated in an ad hoc manner based upon informal requirements documents, a repository of source code, a User's Manual, and some examples of its use. By contrast, our rigorous system design and assurance tests are derived systematically from the client requirements.

Our rigorous systems development method for the aforementioned Dutch election system produced three particularly useful artifacts: (1) a formal specification of the domain model of Dutch elections, (2) a formal architecture description, and (3) a model-based design-by-contract specification of the system.

Producing item (1) revealed over one hundred errors in Dutch election law and the election system that we were asked to integrate with. Catching these errors engendered confidence that the system we created fulfilled the requirements stipulated in law.

Item (2) let the team decompose the development work into three strictly separated subsystems (UI, I/O, and core data types and algorithms), implement and verify those subsystems completely apart from each other (this is called compositional verification), and plug them together at the end of the project, resulting in a system that operated correctly the very first time it was executed. This decoupling permitted the team to parallelize work, avoiding inefficiencies related to inappropriate relationships between subsystems, and let us ensure that the implementation of the system conformed to its architecture, avoiding what is known as architecture drift.

Item (3) helped achieve greater assurance faster than any traditional engineering approach. In particular, our methodology enabled the team to automatically generate unit, subsystem, and integration tests from model-based specifications, saving an enormous amount of time over hand-written tests. Additionally, we provided assurance about the consistency and coverage of those tests against

election law and client requirements because we were able to trace tests all the way from law to code. Finally, we used those specifications to formally verify that the implementation behaved correctly under all possible inputs. We performed this verification using an advanced static analysis technique known as extended static checking, a technology for which Dr. Kiniry and other team members are internationally recognized.

Continuous Validation, Verification, Integration, and Deployment

Development follows a continuous validation, verification, integration, and deployment approach. Each code change is assessed automatically, as soon as it can be, to catch defects as early as possible. Continuous integration is a proven way to reduce development cost and improve productivity. In continuous deployment, code that has passed validation and verification ("V&V") and integration testing is promoted automatically from the main development branch to a deployment staging area that allows all project personnel to access and test the latest working code in a whole-system context.

We will use the external build tool Travis CI, which is configurable and is free of charge to open source projects.

Documentation and commenting is interwoven with development. Beginning with our formal domain models, we lay out the requirements and expectations of each module. We do this both formally (in a language that we can automatically check later) and in plain English. Moving forward, we translate both of these specifications into appropriate constructs (documentation comments, assertions, annotations, type specifications) in our programming language of choice. This enables the automatic generation of PDF and HTML documentation describing each piece of the software and showing relevant code snippets.

We use Git for version control within GitHub and leverage commit hooks (for automatically running testing tools, static checking tools, etc. every time the codebase is changed) and issue tracking. Many of the tools we enumerate in the Developer Instruction documentation have built-in sharing and version control capabilities. We leverage those capabilities and have snapshots of design artifacts captured in our distributed version control system like any other engineering artifact.

When Free & Fair notices a defect, we will log the issue immediately into the issue tracking system within GitHub. If the Colorado Department of State would like direct access to GitHub to log issue, we will provide access; the Colorado Department of State may prefer to notify Free & Fair by email so that Free & Fair can enter the issue into the tracking system. Within a day of entry to the system, each issue will be categorized and assigned to a team member who will be responsible for driving the effort to fix the issue. Each code change that has an effect on an issue will reference that issue, so that progress towards a fix can

be observed as it occurs. We will maintain a policy that an issue can be closed only once the party that raises the issue signs off that the issue has, in fact, been fixed.

All development-related team communication is facilitated by an Asana project and several Slack channels that are accessible to and editable by all team members. Specific Slack channels also serve as the reporting facility for team metrics and the home of software documentation during development activities. We use metrics such as test suite success rate and defect escape rate to monitor code health, adapting our test suites and design review processes to meet goal lines for each metric.

If Colorado wishes to contract us for ongoing upgrades to the system, we will use Ansible or a similar configuration management tool to ensure that environments remain consistent across machines. Note that, since configuration management exists for the convenience of the client, we are comfortable with any variety of configuration management tools (such as CFEngine, Puppet, Chef, etc., as appropriate for the given programming language and deployment platform) and will work with CDOS department to come to a suitable solution.

Finally, we can also package and deliver full snapshots of development environments in virtual machine images. We typically use VirtualBox for such work.

Validation and Verification

Determining whether the system you are creating is the system that a client wants is called *validation*. *Testing* is one means by which to perform validation. Mathematically proving that a system performs exactly as specified under an explicitly stated set of assumptions is called *verification*.

We perform both validation and verification on a regular basis, every night when nightly builds are created and on every merge of new or corrected functionality into the release branch of the development repository. This is standard development practice and minimizes problems that can occur during component integration. It is also part of our continuous integration practice. We discuss our rigorous systems engineering method in greater detail in the Development Process and Methodology document.

Testing and Proving

Testing provides some degree of assurance that a system will behave according to its requirements. In mainstream software engineering, test-driven development, often couched in agile processes, is in vogue and considered a best practice. While we realize testing is important, we do not use testing in the same fashion as other R&D organizations. We are different because, as discussed elsewhere at length, we use a rigorous systems engineering methodology based upon applied formal methods.

Essentially, because we reason about programs and their specifications, rather than hand-writing and hand-maintaining tests that only describe a small fraction of a system's functionality, we formally describe how a system is meant to behave and *prove* (formally, mathematically, mechanically) that the system will always behave that way under all conditions. These correctness proofs give as much assurance as testing every possible state of the system. This field of R&D is known as formal verification; our team includes world leaders in this topic, who have previously been professors and professionals inventing and publishing new concepts, mathematics, tools, and techniques in this area for decades.

Many properties a system should have cannot be tested; security is one of the most noteworthy of these. Certainly, one can search the system for known bad practices or "gotchas" (this is often viewed as "security testing" in mainstream software engineering), but avoiding all of the *known* mistakes one can make says nothing about all of the possible *unknown* mistakes that can introduce security failures in systems.

Despite the level of assurance we can achieve through formal verification, there is still much to be learned from executing a system and examining its behavior under execution, whether in a virtual environment (e.g., virtualization) or in a physical one (across different CPUs, operating systems, etc.). Below we

explain the means by which we test, and how that testing complements formal verification.

System Validation

Free & Fair will ensure that the delivered system meets its intended security, performance, and accuracy (correctness) requirements by applying dynamic checking (classic testing) at three levels: *unit testing*, *functional testing*, and *user interface (UI) testing*.

Unit testing exercises each basic software component of the system to ensure it meets its specification. The specification is obtained as part of the refinement of the high-level system requirements through the design process. Using our tools, most of these tests will be generated automatically. Unit tests provide a low-level indication that the basic components of a system are working as intended and provide an early warning if changes cause requirements and implementation to diverge.

Functional testing exercises the overall functionality of the system. A range of use scenarios is designed to cover the full requirements of the system, including even unlikely combinations of input data. Functional tests can be designed without a full implementation in hand, based on the system requirements and the input data formats; this helps to ensure that the tests accurately represent the system requirements without being influenced by implementation choices.

There are two types of user interface (UI) testing: *usability/accessibility testing* and *UI functional testing*. Usability and accessibility testing will be handled by the Colorado Department of State (CDOS) for this project, in accordance with their processes for such; however, before delivering a version of the system to CDOS for testing, we will run our own set of UI functional tests to ensure that the implemented UI conforms to the design and the product requirements built into the design. UI functional testing is automated using tools that can drive the UI based on test scenarios; the testing is run regularly to be sure that no inadvertent code changes cause the implementation to diverge from the design.

A part of the regular reporting during the project will be the status of test suite development and test suite success, across all of the kinds of dynamic testing described above. Since it is typically not possible to test all possible scenarios, dynamic checking only provides indicative evidence that a system performs as intended; therefore, we will not only report test suite success information, but also indicate what fraction of system behaviors our test suite covers.

System Verification

In addition to system validation through dynamic checking, described above, Free & Fair will use static (formal) verification to prove that key components

implement the system requirements. In contrast to dynamic checking, static verification is performed without executing the software, and provides information about the software's behavior that is independent of particular test scenarios.

To perform static verification, the software implementation and specification are each translated into an equivalent logical representation and automated proof tools are used to ensure that the implementation satisfies the specification. This is the formal variant of testing processes that are known by many names, such as component-based, component-level, and subsystem-level testing. Because our specification and reasoning methods are compositional—we can perform verification on each individual element of the software independently of the others—these techniques also subsume what is normally known as integration testing.

A part of regular reporting during the project will be the status of static verification, including information about exactly what functionality has been statically verified.

Security

This system will be hosted in CDOS data centers. In this document, we will therefore only explicitly address issues not implicitly addressed by this choice. Concretely, due to planned hosting in CDOS data centers, all of the following items are implicitly addressed:

- System fault-tolerance, redundant hosting, and fail-over
- GeoIP blocking, IP whitelisting/blacklisting
- Web application firewalling
- Web application penetration testing and vulnerability scanning
- Distributed denial-of-service prevention
- Anti-malware scanning and other host protections, such as file and configuration integrity monitoring
- Centralized logging
- Network segmentation
- Systems administration and remote access
- We now address the remaining security requirements.

User management and controls

Upon user registration, users will be prompted to create a password conforming to the requirements listed in [CO-RLA-DQ, 13]. For authentication and authorization we will either integrate with the existing CDOS User Management System or implement the following best practices.

Passwords will be salted and stretched using a secure key-derivation function such as PBKDF2 or Argon2, which can be tuned to increase the work factor required for password guess attempts.

Two-factor authentication will be provided via the Time-Based One-Time Password (TOTP) algorithm or another two-factor scheme negotiated with the client.

Policies around password expiry, access attempt controls, and session duration limits will be enforced by the application in accordance with [CO-RLA-DQ, 13] or more recent NIST standards, as negotiated with the client.

User provisioning and password management will be performed by Free & Fair, coordinated with the state.

System operations, security, and privacy

Free & Fair engineers are experts in secure software engineering practices, and regular clients include the Department of Defense and the United States intelligence community. All software will undergo a documented security review

before production release, and will be specifically audited against the OWASP Top 10. In particular, the RLA application will satisfy the Colorado State OIT Secure Applications Coding Standard as described in [TS-CISO-006].

Systems hardening and protection

Free & Fair-delivered systems will be hardened according to best practices. Any sensitive data, including personally-identifiable information (PII), will be stored encrypted. Application-level logging will be implemented using syslog and standard Java logging mechanisms, which will then be aggregated by CDOS-provided centralized logging, using NTP for time synchronization. We will build the system with the security design principles we have used for years for Department of Defense and U.S. intelligence community clients.

Requirements

This document contains the requirements of the specification of the ColoradoRLA system.

History

- Outline and first draft, 5 July 2017 by Joe Kiniry.
- Second draft, 6 July 2017 by Joe Kiniry. Still no refined system requirements above and beyond those in the Documented Quote issued by the Colorado Department of State (DQ).
- Third draft, in progress August 2017, by Stephanie Singer, based on the final Rule 25 promulgated by the Colorado Department of State

Mandatory Requirements

Mandatory Behavioral Requirements

- Ballot manifests and cast vote records (CVRs) will be uploaded to the server via HTTPS.
- The status of uploaded data will be summarized in a state-wide dashboard, along with information on which counties have not yet uploaded their CVRs, and uploads that have formatting or content issues. The status of data, and results as audits are performed, will be provided for each contest to be audited.
- Random selections of ballots for performing a ballot-level comparison risk limiting audit will be automatically generated based on the provided random seed using the SHA-256-based pseudo-random number generator specified in the DQ, as well as the computed contest margins and other indicated parameters including any discrepancies found.
- A county view of the audit will display information on the progress of each audited contest in the county, with a summary of discrepancies.
- We will tailor view/edit permissions for each screen of information to users with appropriate authorizations as defined by the state.
- Public access to appropriate data and reports will be provided in standard file formats.

Mandatory Non-Behavioral Requirements

We summarize our approaches to three critical non-behavioral properties of this system below: how we achieve *fault tolerance*, how we perform *synchronization*, and some reflections on the *dynamism of the UI and user experience (UX)*.

Fault tolerance

We have built many fault tolerant distributed systems over the years. For example, Dr. Kiniry was co-architect of Sprint's Internet Service Provider product (what later became a part of Earthlink) in 1995. While the underlying platforms and technology have evolved several times since then, the underlying principles remain the same.

For this particular application, we propose a two server deployment, preferably in separate locations which have independent power subsystems and network backbones. Each server will have two power supplies and two network cards, which should be on separate, independent subnets. Servers will have hot-swappable SSDs in a RAID 5 configuration for local data redundancy and fault tolerance.

Synchronization

We can design a distributed synchronization protocol in either a primary/secondary architecture (with support for dynamic failover in the case of network or system unavailability) or in a peer-to-peer configuration, where inbound requests can go to either server using DNS load balancing. We can also use a distributed synchronization mechanism already integrated into the client-selected backend database system, if that is an appropriate choice. The decision of what synchronization protocol to use will be made in consultation with the client.

UI and UX Dynamism

Rather than simply create a plain-old-HTML front-end, our UI and UX will use rich JavaScript UI libraries to create a browser-based user experience that feels like a modern application one might find in any of the mainstream online app stores. Our UX expert will work with the client to ensure that the UI's dynamism and presentation facilitates OpenRLA's critical users, election officials running audits.

Secondary Requirements

- This system will be hosted in the CDOS data center, which will support failover, as described in the Documented Quote [CO-RLA-DQ, p. 16].

- Our system includes all necessary features for fault tolerance summarized earlier, including a redundant standby server with separate network cards and power supplies, dual CPUs, and hard drives in a RAID 5 configuration.
- We expect regular backups to be configured to use CDOS Disaster Recovery facility.
- Ensure business continuity of the system with at least four nines (99.99%) availability.

System Specification

The formal system of the RLA Tool is specified in a Literate PVS specification in `specs/pvs/corla.pvs`. We will integrate the two documents for final system delivery. See the working draft, generated from that formal specification, in our GitHub repository.

Domain Analysis and Engineering

To be written.

System Architecture

To be written.

Behavioral Specification

To be written.

Non-Behavioral Specification

To be written.

Deployment Considerations

The ColoradoRLA system is conceptually a 3-tier system (browser-based TypeScript client, server-side Java application, database). However, because it is relatively lightweight, it can be deployed in many configurations ranging from a single machine hosting all 3 tiers to multiple redundant servers hosting each tier. Because the system stores state only in the database (and on the user's local machine, in the browser-based client), load can be balanced across the client layer and the Java application layer using basic techniques such as DNS round-robin and load-balancing front-end servers. Balancing load across the database layer, however, requires ensuring database consistency in one of various ways.

High Availability

The ColoradoRLA system deployment for the Colorado Department of State has a high availability requirement. Achieving high availability for the client-side and server-side applications through the use of multiple servers is straightforward. For the database, high availability requires some form of replication that ensures data consistency.

We are using the Postgres DBMS for the database layer. Postgres supports several different types of replication. For the CDOS deployment, our initial recommendation is to deploy two Postgres servers (one primary server and one replica) and use Postgres's integrated streaming replication functionality, with synchronous commits enabled to achieve group-safe and 1-safe replication. In this configuration, the only possibility for data loss is if both the primary server and the replica crash simultaneously, *and* the database on the primary server is corrupted at the same time. We could opt for a stronger durability guarantee, achieving 2-safe replication (the only possibility for data loss is if both servers crash simultaneously in a way that corrupts both databases), but that would cause longer transaction times. See https://www.postgresql.org/docs/current/static/warm-standby.html#SYNCHRONOUS-REPLICATION and http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1
for more information.

Project Members

Dr. Joseph Kiniry is Chief Scientist and CEO of Free & Fair. Prior to working for Free & Fair, Dr. Kiniry provided commercial and public consultancy services to several governments on matters relating to elections, their technology, security, processes, and verifiability. He has worked on election systems for fifteen years; has audited the security, correctness, and reliability of numerous physical and Internet-based voting systems; and has developed high-assurance prototypes and products of several election technologies (including, but not limited to, tallying, auditing, voting, ballot marking, and e-poll book (EPB) systems).

Dr. Kiniry has formally advised four national governments (the USA, The Netherlands, the Republic of Ireland, and Denmark) on matters relating to digital elections and has testified before two parliaments. He has also provided informal input and advice to the governments of Norway, Estonia, and the United States. He co-founded and co-ran a multi-year research project on digital elections (the DemTech project) and has supervised numerous B.S., M.S., and Ph.D. theses focusing on election technologies. His research group developed several high-assurance peer-reviewed election software systems, including a tally system used in binding European elections for The Netherlands in 2004 and an EPB system used in Danish national elections in 2012. Dr. Kiniry has served as a Principal Investigator on research projects for the European Union Council, various Department of Defense branches, the National Science Foundation, and several national funding agencies in Ireland, The Netherlands, and Denmark. He has also started and run a half dozen technology firms and has held tenured positions at four university in three countries. He holds five advanced degrees, including a Ph.D. from the California Institute of Technology.

Joe Ranweiler is a software engineering consultant for Free & Fair. He has over 5 years of professional experience building software, spanning research and development, data engineering, and commercially-deployed web applications. Joe helped write Free & Fair's end-to-end verifiable voting system demonstrator, including its core cryptographic components. He is a regular open-source software contributor and was recently the technical lead on Free & Fair's OpenRLA risk-limiting audit system prototype, which was built using modern web application technologies. Joe has a B.S. in Mathematics from Arizona State University.

Neal McBurnett has been developing open source software related to election audits for over a decade, and worked as a software developer for tools and the Internet as a Distinguished Member of Technical Staff at Bell Labs for two decades before that. He consulted with the Colorado Secretary of State on the Colorado Risk-Limiting Audit project, and is a member of the team that worked with County Clerk Dana DeBeauvoir in Travis County, TX on the design and RFP for STAR-Vote, a novel voting system supporting end-to-end and risk-limiting audits. He served as vice-chair of the IEEE P1622 standards committee on a common data format for elections, and continues to participate in the U.S. Election Assistance Commission's VVSG-Interoperability

Working Group, developing standards for Cast Vote Records and related formats. Using his open source web-based ElectionAudits software, Boulder County, CO performed nationally-recognized audits in 2008 and 2010. He also audited the groundbreaking open source Scantegrity end-to-end election in Takoma Park, MD in 2011. Mr. McBurnett was a major contributor to "Principles and Best Practices for Post-Election Audits" (September 2008) and the 2010 American Statistical Association statement on Risk-Limiting Small Batch Audits. He has participated actively in election processes since 2002 as an observer, election official, auditor and public witness. He is an active participant in the Election Verification Network. He holds an M.S. in Computer Science from the University of California at Berkeley, and a B.S. in Computer Science from Brown University.

Dr. Daniel Zimmerman, the Technology Lead at Free & Fair, has extensive experience in formal methods, high-assurance software engineering, concurrent and distributed systems, and foundations of computer science. He taught computer science at multiple universities for over a decade. In industry, he has worked primarily in the areas of rigorous software engineering and verifiable election technology. He holds three advanced degrees, including a Ph.D., all from the California Institute of Technology.

Dr. Joey Dodds has focused mainly on research and development facilitating correctness proofs for a variety of programs, including cryptographic algorithms. At Free & Fair, Dodds has implemented both a tabulator and a risk-limiting audit system and written formal specifications for both. He also fully proved the correctness of the tabulator. He is a key participant in the verification of Amazon's s2n library, responsible for both the verification of the library and implementing a system to automatically report metrics about the progress of the project to Amazon's upper management. He holds a Ph.D. from Princeton University and holds two other advanced degrees.

Dr. Stephanie Singer has developed web-based applications querying relational databases to make customized reports of election results available to the general public. As a member of the Philadelphia County Board of Elections in Pennsylvania, she oversaw the creation and deployment of a modern voter-facing election website. She also held a tenured position in mathematics at Haverford College for over a decade and earned several degrees, including a Ph.D. in mathematics from NYU.

Mike Kiniry is a communication expert, with backgrounds in radio, print, and photojournalism, who specializes in clearly communicating complicated concepts. He spent nearly a decade as a public radio reporter, producer, and host and has been a freelance writer and photographer for the past 15 years. Mike is also a videographer and editor, and is the Election Verification Network's dedicated videographer/media producer.

Morgan Miller is an experienced User Experience (UX) professional with a deep background in scientific research. She is currently a User Experience Architect for Morgan Miller UX, LLC, where she leads teams through a UX

discovery, architecture, and research process; designs and executes research studies; synthesizes research data to create actionable recommendations; and builds information architecture including taxonomy, sitemaps, and wireframes. She has done work for Overseas Vote Foundation, Intel, Mozilla Foundation, BMC Software, Esri, World Wildlife Fund, Nike, Moda, Providence Health, and Cambia Health. She earned a B.A. in Mathematics from Reed College and an M.S. in Computer Science from the University of Lugano, Switzerland, where she was a cryptography researcher.

Glossary

See also the working documents at VVSG-Interoperability Voting Glossary and the glossary in: "Risk-Limiting Post-Election Audits: Why and How"

- **RLA Tool** A computer system for conducting a Risk Limiting Audit
- **business interruption** - Any event that disrupts Contractor's ability to complete the Work for a period of time, and may include, but is not limited to a Disaster, power outage, strike, loss of necessary personnel or computer virus.
- **closeout period** - The period beginning on the earlier of 90 days prior to the end of the last Extension Term or notice by the State of its decision to not exercise its option for an Extension Term, and ending on the day that the Department has accepted the final deliverable for the Closeout Period, as determined in the Department-approved and updated Closeout Plan, and has determined that the closeout is complete.
- **deliverable** - Any tangible or intangible object produced by Contractor as a result of the work that is intended to be delivered to the State, regardless of whether the object is specifically described or called out as a "Deliverable" or not.
- **disaster** - An event that makes it impossible for Contractor to perform the Work out of its regular facility or facilities, and may include, but is not limited to, natural disasters, fire or terrorist attacks.
- **key personnel** - The position or positions that are specifically designated as such in this Contract.
- **operational start date** - When the State authorizes Contractor to begin fulfilling its obligations under the Contract.
- **other personnel** - Individuals and Subcontractors, in addition to Key Personnel, assigned to positions to complete tasks associated with the Work.
- **start-up period** - The period starting on the Effective Date and ending on the Operational Start Date.
- **ballot manifest** - A document that describes how ballot cards are organized and stored, and relates a Cast Vote Records to the physical location in which the tabulated ballot card is stored. The ballot manifest specifies the physical location of a ballot card to allow staff to find the specific ballot card represented by a given CVR. A ballot manifest will contain the following information: county ID, tabulator ID, batch ID, the number of ballot cards in each batch, and the storage location where the batch is secured following tabulation. A sample ballot manifest is provided at `manifest-dq.csv`

- **cast vote record (CVR)** - An electronic record indicating how the marks on a ballot card were interpreted as votes. May be created by a scanner or DRE, or manually during an audit. Sample CVRs in Dominion's format are in `test/dominion-2017-CVR_Export_20170310104116.csv`. See also VVSG-Interoperability CVR Subgroup.
- **contest** - Any decision to be made by voters in an election, such as a partisan or nonpartisan candidate race, or a ballot measure. Ex: Jane Doe for Colorado Secretary of State. Each option for the voter is called a *choice*.
- **choice** Any possible outcome of a Contest. In a Contest to determine who will fill a certain office, each choice is a person, called a candidate for the office. In a ballot question contest, the choices are "Yes" and "No".
- **coordinated election** - Coordinated Elections occur on the first Tuesday of November in odd-numbered years. If the Secretary of State certifies at least one statewide ballot measure to the counties, every county will conduct the Coordinated Election, and the vast majority of counties will include additional local ballot content in the election. If the Secretary of State does not certify at least one statewide ballot measure to the counties, then only those counties to which local political subdivisions certify ballot content will conduct a Coordinated Election in that year.
- **county administrator** - The designated representative(s) of each county clerk and recorder who possesses RLA administrative user privileges sufficient to upload a cast vote record and ballot manifest for the county.
- **contest name** - The title of a contest.
- **election day** - The final day on which voters can cast a ballot in a State Primary Election, Presidential Primary Election, Coordinated Election, or General Election.
- **offeror** - A vendor that submits a responsible bid for this Documented Quote.
- **pseudo-random number generator** - A random number generator application that is further explained at http://statistics.berkeley.edu/~stark/Java/Html/sha256Rand.htm Test data is available at https://github.com/cjerdonek/rivest-sampler-tests
- **random seed** - A random seed (or seed state, or just seed) is data, such as a number, vector or string, used to initialize a pseudorandom number generator.
- **responsible bid** - A bid from a vendor that can responsibly (i.e. is reasonably able and qualified) do the work stated in the solicitation.
- **risk-limiting audit (RLA)** - A procedure for manually checking a sample of ballot (cards) (or other voter-verifiable records) that is guaranteed to have a large, pre-specified chance of correcting the

reported outcome if the reported outcome is wrong. (An outcome is wrong if it disagrees with the outcome that a full hand count would show.) One paper describing risk-limiting audits is located at https://www.stat.berkeley.edu/~stark/Preprints/gentle12.pdf.

- **state administrator** - The designated person who possesses RLA administrative user privileges to perform administrative tasks.
- **tabulation** - Aggregation of tallies of interpretations of voter choices into election results. @review NIST Election Modeling group separates interpretation, tally and tabulation.
- **tabulated ballots** - Paper ballot cards that have been scanned on a ballot scanning device, and the voter markings on which have been interpreted by the voting system as valid votes, undervotes, or overvotes. Tabulated ballots may be duplicates of original ballots. @review this means ballots counted by hand weren't "tabulated". Is that what we want to say?
- **two-factor authentication** - Defined as two out of the three following requirements:
- Something you have (Examples: token code, grid card)
- Something you know (Example: passwords)
- Something you are (Example: biometrics)
- **ENR system** An Election Night Reporting system, a computer system enabling publication of election results starting on election night, and continuing through the end of certification.
- **reported outcome** - The set of contest winners published by the ENR system.
- **calculated outcome** - The set of contest winners according to the CVRs that are being audited.
- **wrong outcome** - When the reported outcome for a given contest does not match the outcome that a full hand count of the paper ballots would show. This can happen due to equipment failures, adjudication errors, and other reasons.
- **full hand count** - TBD along these lines: A procedure for determining the correct outcome of a contest, suitable for use in an RLA. It involves a tabulation of the votes for each choice in a contest which involves manual interpretation of each ballot, and may involve verifiable machine assist with checking the counts. See one suggested procedure at Branscomb full hand countproposal. Cf. *recount*.
- **recount** TBD, a procedure under Colorado law that happens *after* certification if the margin is too tight, or if a candidate requests it. A recount

doesn't have to involve manual interpretation of each ballot. Cf. *full hand count*.

- **overstatement of the margin** An error whose correction reduces the margin https://www.stat.berkeley.edu/~stark/Preprints/evidenceVote12.pdf
- **understatement of the margin** An error whose correction increases the margin https://www.stat.berkeley.edu/~stark/Preprints/evidenceVote12.pdf
- **evidence-based elections** - An approach to achieving election integrity in which each election provides affirmative evidence that the reported outcomes actually reflect how people voted. This is done via software-independent voting systems, compliance audits and risk-limiting audits. An alternative to certifying voting equipment and hoping that it functions properly in real elections. See also *resilient canvass framework*. See Evidence-Based Elections - P.B. Stark and D.A. Wagner
- **resilient canvass framework** - A fault-tolerant approach to conducting elections that gives strong evidence that the reported outcome is correct, or reports that the evidence is not convincing. See also *evidence-based elections*.
- **compliance audit** - An audit which checks that the audit trail is sufficiently complete and accurate to tell who won. Generally includes poll book accounting, ballot accounting, chain of custody checks, security checks, signature verification audits, voter registration record auditing, etc. Related terms include election canvass, ballot reconciliation. See https://www.stat.berkeley.edu/~stark/Preprints/evidenceVote12.pdf
- **audit board** - Given a County, a group of electors in the county nominated by the major party chairpersons, which carries out an audit, with the assistance of the designated election official, members of his or her staff, and other duly appointed election judges.
- **audit** A process by which the performance or outcome of a system is verified. @review TBD May include ballot tabulation audits, compliance audits, ...
- **RLA** Risk Limiting Audit
- **Risk Limiting Audit** An Audit designed to reduce the statistical probablility that a wrong election winner was determined by an interpretation and tabulation system.
- **ballot tabulation audits** An Audit of a vote-tabulation system. @review TBD. including risk-limiting audits, opportunistic audits, bayesian audits, fixed-percentage audits, etc. @review if we honor the NIST Election Modeling Group distinction between interpretation and tabulation, this definition describes a "ballot interpretation and tabulation audit"

- **opportunistic audit** - An auditing technique designed to efficiently generate evidence for additional contests in a ballot-level audit. A significant part of the effort in doing a risk-limiting audit involves physically retrieving the ballots selected for audit. While doing the manual tabulation and entering the data for the contests on that ballot which are subject to strict risk limits, it is possible to "opportunistically" do the same thing for other contests that are observed on the same ballot, producing evidence about them for little additional effort. These are called "opportunistic contests". If an opportunistic contest achieves a risk limit, it can be "settled", and when it appears on subsequent ballots during the audit, it need not be tabulated. TBD: discuss need to consider possibility of sampling bias when evaluating and reporting, considerations for possible escalation, etc.
- **mandatory contest** - A Contest in a Risk Limiting Audit which is subject to a Risk Limit and hence is factored in to the sampling calculations.
- **opportunistic contest** - A contest to be audited opportunistically.
- **Risk Limit Goal** In a Risk-Limiting Audit, each Contest has a Risk-Limit Goal, namely, the acceptable risk that an incorrect outcome will escape notice.
- **Dynamic Risk** In a Risk-Limiting Audit in progress, for any specific Contest Under Audit, at any given time the Dynamic Risk is the probability, based on Ballots audited so far, that an incorrect outcome of the Contest as escaped notice.
- **Under Audit** A Contest that has been chosen for audit.
- **active contest** At any given time, a Contest Under Audit whose Dynamic Risk exceeds the Risk Limit Goal @review this is just a guess @review what's the origin of this term? Couldn't find it in Stark or Lindeman/Stark.
- **settled contest** At any given time, a Contest Under Audit whose Dynamic Risk is less than or equal to the Risk Limit Goal @review TBD involving having achieved Risk Limit Goal. Note need to ensure that calculations take into account the way the samples were selected, in case any samples were taken in a stratified manner or taken non-uniformly in order to target non-county-wide contests.
- **uncontested contest** A Contest for which the number of choices is less than or equal to the number of winners @review Consider a contest with three winners where each voter can vote for two. Are we OK calling that an "uncontested contest"? Or a contest (such as Republican Committeeperson in Philadelphia) with a minimum number of votes required for a write-in candidate? And in jurisdictions without qualification or other restrictions on counting write-in votes, how can we be sure that any contest is uncontested?
- **bayesian audits** @review Neal McBurnett

- **voting method** TBD
- **electoral system** the method used to calculate the number of elected positions in government that individuals and parties are awarded after elections.
- **ballot** a list of contests and, for each contest in the list, a list of choices, in a form that allows a person to record choices and allows a person or a computer (or both) to read recorded choices. Each ballot is composed of one or more ballot cards, and each ballot card has a unique ID.
- **ballot card** a single physical page of a ballot.
- **margin** Given a contest and two choices in that contest, the numerical difference between the choice that got more votes and the choice that got fewer votes.
- **hash function** TBD, mentioning specifically SHA-256
- **RLA software** The software component of an RLA Tool.
- **ballot storage bin** A physical container for a set of paper ballot cards.
- **batch** A set of Ballot Cards which has a numeric id and a size (the number of Ballot Cards contained in the batch).
- **batch id** Each Batch has a unique Ballot Identifier.
- **batch size** the size of a batch, and virtually all batches are identically sized.
- **chain of custody** Given an item (e.g., Marked Ballots, Unmarked Ballots, Ballot Cards) in need of security over a certain time period, the chain of custody is the sequence of people, organizations or locations where the item remains secured over the given time period.
- **county** (in the US) a political and administrative division of a state, providing certain local governmental services, including conducting elections.
- **scanner** A machine that can take Paper Ballots as input and whose output is a CVR for each Paper Ballot.
- **imprinted ballot** - A Paper Ballot Card on which a unique Ballot Card identifier has been imprinted in order to facilitate a Ballot Card-level audit. The unique Ballot Card identifier might for instance include a unique Batch Identifier and the sequence of the Ballot Card within the Batch, or it might be a new identifier which is also included in the CVR. Imprinting should be done after the ballot card is cast and with care taken to avoid causing anonymity problems.
- **ballot order** A specific ordering of a set of Ballot Cards.
- **Secretary of State (SOS)** In most states in the United States, an office of government defined in the state constitution.

- **Department of State (DOS)** An agency of government which, in most states of the United States of America, is charged with oversight of state elections.
- **audit report** TBD
- **SOS audit form** TBD
- **ballot certification** TBD
- **UOCAVA voter** A person entitled to cast a UOCAVA Ballot
- **UOCAVA ballot** A certain type of absentee ballot prescribed by the federal Uniformed and Overseas Citizens Absentee Voting Act and codified in 52 USC Ch. 203.
- **mail ballot** A Paper Ballot that may be cast by physical delivery to the Board of Elections, usually via the US Postal Service.
- **election canvass** The process of counting and verifying, or "canvassing," the various precinct votes and making determinations on the election results.
- **canvass board** The body which conducts the election canvass and certifies the winners of county and local offices, as well as certifies county vote totals for state and federal offices that extend beyond the county limits.
- **post-election (historical, random) audit** Traditional post-election audits typically entail randomly selecting a few precincts or voting machines, and checking the associated results with a hand count of the paper ballots. While these audits do provide some evidence that the machines have correctly interpreted voters' intent, they audit only a few samples, often using an artificially created subset of results, rather than the actual election results.
- **county clerk** An office of County government, established by the Colorado Constitution (Article XIV, Section 8) and responsible for conducting elections in the County.
- **sample size** TBD including **initial sample size**
- **equipment** TBD
- **VVPAT** - A Voter-Verifiable Paper Trail consists of an audit trail of Voter-Verifiable Paper Records (VVPRs). Elections which produce a VVPAT allow an audit to gather evidence which the voter had an opportunity to verify. The VVPRs may appear in a continuous roll of paper used to provide auditability for a DRE.
- **VVPR** - A Voter-Verifiable Paper Record, also known as a Voter-Verifiable Paper Ballot (VVPB). 'Voter-verified' refers to the fact that the voter is given the opportunity to verify that the choices indicated on the paper

record correspond to the choices that the voter has made in casting the ballot. Risk-limiting audits require VVPRs.

- **non-voter-verifiable ballot** (NVVB) - A ballot for which there is no auditable VVPR. For example a ballot sent via an online ballot return system or email, for which the voter has not returned a matching voter verifiable paper ballot. AKA digital ballot.
- **phantom ballot** An entry in the Ballot Manifest for which there is no corresponding Paper Ballot Card. Phantom ballots can represent pancies between the manifest and the actual paper ballot card batches. A manifest with a batch of purely phantom ballots can also be used to represent the maximum number of possibly late-tabluation ballots.
- **late-tabulation ballot** - A ballot which is tabulated after the CVR report and manifest are generated, but before the canvass is finished.
- **duplicated ballot** A Ballot marked by an Election Official by copying voter choices from another Ballot.
- **original ballot** A Ballot from which a Duplicated Ballot has been created.
- **DRE** A voting system whose primary record of voter intent is an electronic record created by a voter's physical interaction with a voting machine.
- **votes allowed** Given a Contest, the maximum number of choices a voter may legitimately select in that Contest.
- **overvote** Given a Marked Ballot and a Contest, a selection of more choices than the Votes Allowed for that Contest.
- **stray mark** A Ballot Mark that does not carry any information about voter intent.
- **damage** TBD
- **undervote** Given a Marked Ballot and a Contest, a selection of fewer choices than the Votes Allowed for that Contest.
- **risk limit** - The pre-specified minimum chance of requiring a full hand count if the outcome of a full hand count would differ from the reported tabulation outcome.
- **voting system** TBD
- **Dr. Philip Stark** Associate Dean, Division of Mathematical and hysical Sciences, Professor of Statistics, University of California. https://www.stat.berkeley.edu/~stark/
- **Dr. Mark Lindeman** Mark Lindeman is a political scientist who studies public opinion and elections. He presently lectures at Columbia University

in quantitative methods, and led the revision of Carroll Glynn et al.'s multi-disciplinary textbook Public Opinion. https://electionverification.org/aee-statistics-and-auditing/

- **Dr. Ron Rivest** Professor Rivest is an Institute Professor at MIT. He joined MIT in 1974 as a faculty member in the Department of Electrical Engineering and Computer Science. He is a member of MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL), a member of the lab's Theory of Computation Group and a founder of its Cryptography and Information Security Group. He is a co-author (with Cormen, Leiserson, and Stein) of the text, Introduction to Algorithms. He is also a founder of RSA Data Security, now named RSA Security (the security division of EMC), Versign, and Peppercoin. Professor Rivest has research interests in cryptography, computer and network security, electronic voting, and algorithms. http://people.csail.mit.edu/rivest/
- **Colorado House Bill 09-1335** an act of the Colorado Legislature concerning requirements for voting equipment.
- **EAC** An independent, bipartisan commission of the United States Federal Government, established by the Help America Vote Act of 2002.
- **Clear Ballot Group** An election technology company. https://www.clearballot.com/
- **Clear Ballot ClearCount** Clear Ballot Group's browser-based central count tabulation, consolidation and reporting system.
- **OpenCount** OpenCount is a system that can interpret scanned paper ballots and interpret them into cast vote records. It can understand some of the existing proprietary file formats for other vendors' equipment, and can semi-automatically figure out the shape and nature of a ballot with a little help from an elections official. OpenCount was originally designed and implemented at Berkeley under the guidance of Prof. David Wagner.
- **Dominion** Dominion Voting Systems is an election technology company. http://www.dominionvoting.com/
- **Ballot Retrieval** @review morganmillerux the process of getting the ballots seems like a specific point of concern and merits being called out.

Bibliography

Introduction to be written. We will be creating a new repository for the bibliography. The contents will be copied from the OpenRLA repository and organized here.

Risk-Limiting Audits

To be written.

Jurisdiction Sources

To be written.