Verified but Not Secure

Open Problems in Deployed Verifiable Voting Systems

Thomas Haines · Jarrod Rose

Australian National University

FMOZ · 2026

Verifiable e-voting, in one slide

Individual Verifiability

The voter can check that their own ballot was cast as intended and recorded as cast.

Universal Verifiability

Anyone can check that the announced result is the correct tally of the recorded ballots.

Voting Secrecy

No-one learns more than the result — even a corrupt subset of authorities can't link voters to votes.

The thesis of the talk: the audited core delivers these. The deployed system around it doesn't always.

Finally (mandating) verification

Swiss Ordinance on Electronic Voting

2.14.1

A symbolic and a cryptographic proof of compliance must demonstrate that the cryptographic protocol meets the requirements in Numbers 2.1–2.12.

The symbolic proofs are done in Proverif.
The computational proofs are not currently worth the paper they are written on.

Belenios

Cheval, Cortier, Debant — *Election Verifiability with ProVerif*, CSF 2023. Provides automated symbolic proofs of individual and universal verifiability for Belenios

Baloglu, Bursuc, Mauw, Pang — *Election Verifiability Revisited: Automated Security Proofs and Attacks on Helios and Belenios*, CSF 2021

Cortier, Drăgan, Dupressoir, Warinschi — *Machine-Checked Proofs for Electronic Voting: Privacy and Verifiability for Belenios*, CSF 2018.

And yet

10

weaknesses found

across two mature, well-audited systems

6

exploitable

against the main security properties

0

in the audited core

every finding sits outside it

SwissPost · Belenios — the most-scrutinised verifiable systems we have

The audit safety net

What current methodology covers — and where this talk lives

All ten findings live here.

Errors in the specification & primitives are increasingly caught.
What slips through is how the deployed system handles input/output and state — the wiring around the audited core.

Why SwissPost and Belenios

Choosing mature systems is the point

SwissPost

- Regulated under Swiss Federal Chancellery Ordinance
- Mandated audit before each release
- Microservice architecture; multiple concurrent ballot boxes
- Bug bounty: 70–230k€ for IV/UV, 40–50k€ for privacy
- **Our 8 findings would have been worth 360k–1M€**

Belenios

- Built on Helios; long-standing, openly developed
- Widely deployed for academic and public elections
- Basis for FLEP (French legislative elections)
- Symbolic and computational proofs of security
- *Fewer requirements, self-imposed rather than legislated*

If anywhere is safe, it should be here.

Three categories of vulnerability

We focus on the right two

1. Spec & Primitives

Bugs in the specification, or in implementations that deviate from it.

Excluded — the field already focuses here.

2. Input / Output

Code handling I/O to the core protocol misbehaves under adversarial input.

3. State

Core protocol invoked at times or on inputs the security model never envisioned.

3.1 differing views
3.2 temporal state
3.3 non-temporal state

The ten findings

Eight on SwissPost, two on Belenios · six exploitable

ID	Weakness	Impact	Cat.
V1	API free for all	Extra voter credentials	3.2
V2	Shuffling the Shuffles	*Universal verifiability breach*	2
V3	Spoofing Elections	*Universal verifiability breach*	3.1
V4	Triggering an Early Tally	*Privacy breach (later detected)*	3.2
V5	Messing with Zip Files	Invalidates audit claims	2
V6	Extra Voters	Extra voter credentials	3.3
V7	Permuting IDs and Keys	*IV + privacy breach, DoS*	2, 3.3
V8	No Filtering	Universal verifiability breach	1.1
V9	Audit too Late (Belenios)	Privacy breach (later detected)	1.1, 3
V10	Ghost Voters (Belenios)	Voters issued invalid credentials	3.3

Italic impact = breaks claimed security property.

V2 · Shuffling the Shuffles

CAT 2 · I/O

SwissPost mixnet across two ballot boxes — setup

Online (CC1–CC4)

Shuffle + partial-decrypt

→

Tally (offline)

Final shuffle + decrypt

Two ways to identify a shuffle payload

Online components: load by payload contents (group by the ballot-box ID inside the payload)

Tally component: loads by filename (group by the file's name on disk)

Same data, two different identity schemes. That gap is the vulnerability.

V2 · The swap

CAT 2 · I/O

Adversary lays out the filesystem to fool the offline tally

On-disk layout (adversarial)

Folder BB1/

payloads tagged BB2 inside

Folder BB2/

payloads tagged BB1 inside

Online verifier

groups by ID-in-payload → consistent. All proofs verify.

Tally component

groups by filename → walks into the wrong box.

Key shares

CC2+CC3+CC4 pre-arranged to cancel; CC1's share is honest. Ciphertexts decrypt cleanly — in the wrong election.

Result

BB1's announced plaintexts are BB2's votes, and vice versa.

Every local check passes. The bug lives between locally verified and globally guaranteed.

V3 · Spoofing Elections

CAT 3.1 · STATE

Two participants. Two different elections. Nobody notices.

What the verifier sees

- Election config: N ballot boxes
- Receives signed verification data for exactly N
- Checks correctness → passes

What the CCs see

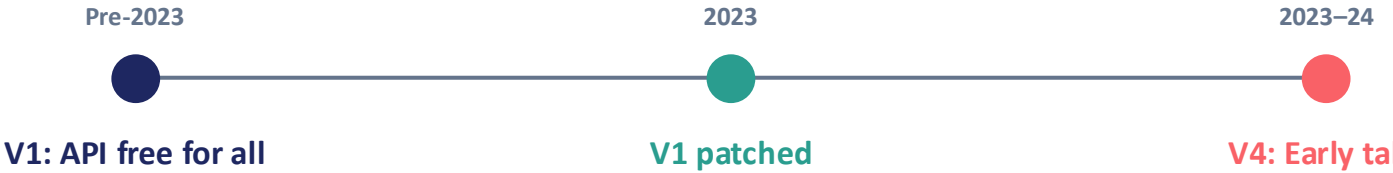
- Election config: N + K ballot boxes
- Process all N + K (including the K “extra”)
- No-one cross-checks the configs match

Ballots in the K extra boxes are silently uncounted. Worst case: every honest ballot discarded, adversarial ones substituted.

V1 → V4 · The same class, twice

CAT 3.2 · PHASE

Why per-bug patching is not enough



Phase enforcement was treated as a per-endpoint fix. New code, new endpoint, same class of bug.
This is why category-level thinking matters — not just patching V1, V2, V3 … one at a time.

The ProVerif auditor vs. the deployed verifier

The universal-verifiability theorem rests on this 8-line process

Universal-Verifiability-2CCMs.pv

```
let Auditor =
  in(CCM2Channel, (E, Mix1, MixProof1));
  in(c,           (Mix2, MixProof2));
  if VerifP11(...) = true
     || VerifP12(...) = true then
  if VerifP2(...) = true then
  out(c, (E, Mix1, Mix2, MixProof2));
  event HappyAuditor(E, Mix1, Mix2).
```

What the deployed verifier actually has to do

- Load and parse zip files
- Reconcile setup-phase vs. tally-phase context
- Check filename ↔ content consistency (V2)
- Check election configuration matches across components (V3)
- Handle multiple concurrent ballot boxes

Why audits and proofs miss them

Category 2 · I/O

- Honest implementations produce inputs the others accept.
- Bug only surfaces under inputs an honest party would never construct.
- Standard testing and review optimise the honest path.
- Missing professional paranoia in implementers.

Category 3 · State

- Specs encode phase / consistency constraints in prose, not state machines.
- Deployment complexity (microservices, batching, concurrent ballot boxes) lives outside the formal model.
- Implementers infer obligations from narrative; mismatches go silent.

ProVerif, Tamarin, EasyCrypt missed every one — the formal model never represented the actual structure.

Easy wins

Make the implicit assumptions of the security analysis explicit and testable

API input checklists

For each endpoint, auditors enumerate the inputs a corrupt coalition could supply and check each is handled safely.

Systematic phase enforcement

Per-endpoint pre/post-conditions; model-check, fuzz, property-test. Applied to all APIs, not patched one at a time.

Consistency against prior state

Every honest component validates input against its own accumulated view of the election.

Document implicit assumptions

Treat them as a structured part of the spec — not as folklore in the narrative.

None of these require changing the underlying formal definitions.

Possible deeper solutions

Extend security definitions to capture state (and maybe I/O) explicitly

Possible in principle. Definitions become so complex they're hard to reason about on paper, and harder to mechanise.

Refine from idealised model to deployed variant

A principled refinement discipline would help. We don't have one that scales to microservices + batching + concurrent ballot boxes.

Push more verification into the auditor

Helps for the things we already know about. Doesn't close the class — the next implicit assumption is, by definition, the one we haven't named.

Two directions

01

The research community

Does your part of formal methods already have techniques which might combine with symbolic verification to close some of these gaps?

02

Vendors and stakeholders

Check your systems for input/output and state vulnerabilities specifically. The audited core is not enough.

Thomas Haines · thomas.haines@anu.edu.au · eprint.iacr.org/2025/2033