Auditing the Swiss Post E-voting System: An Architectural Perspective

Bryan Ford*

April 4, 2022

Contents

*The author is a faculty member of EPFL, the École Polytechnique Fédérale de Lausanne. The work described in this report was performed in the author's individual capacity, however, as an ancilliary activity governed by LEX 4.1.1 and the Ordinance on the EPF faculty (RS 172.220.113.40)). It should go without saying that all opinions expressed in this report are solely the author's, and do not in any way reflect institutional positions of or endorsements by EPFL or the ETH domain.

1 Overview of the audit

Switzerland is one of the few countries globally that has a national program for electronic voting (E-voting), which has been evolving in several stages for well over a decade. For the past several years, the program's most recent stage has focused on introducing strong cryptographic verifiability into the system, together with security features such as trust splitting and a software implementation open to public inspection. Although multiple system providers have participated in the E-voting program in the past, currently the only active provider is the SwissPost.

This report represents the author's analysis of the Swiss Post's E-voting system as part of an audit that commenced in 2021. This author's primary role in this audit was to examine the overall architecture of the system from a high-level systems-security perspective, and to study how all the elements of the system fit together to implement an "end-to-end" electronic voting process.

1.1 Methodology

The author formally participated in all three "scopes" of the audit: namely of (1) the cryptographic protocol, (2) the software implementation, and (3) the infrastructure. However, the author never attempted, nor would have been able to, perform a *complete*, detailed audit of all three of these scopes. In particular, the author has attempted neither to verify all the details of the protocol and cryptographic proofs in scope 1, nor to examine the system's source code thoroughly in scope 2, nor to check every aspect of the E-voting infrastructure for compliance in scope 3. The author's participation in all three scopes served instead to feed the high-level architectural perspective this report takes with relevant information from all scopes and aspects of the system, and to examine the system across audit scopes and levels of abstraction.

In accordance with this focus on taking a high-level architectural perspective, this report starts by examining broad architectural issues and concerns in Section 2. Subsequent sections focus on observations that fit primarily into particular audit scopes.

1.2 Important caveats

Several important caveats apply to this report:

- The report may well contain accidental mistakes or incorrect judgments based on the author's incomplete understanding of the system in all its details.
- The author's analysis is substantially based on reference documents summarized below, all of which were working drafts under continuing evolution. The observations in this report might, or might not, be valid with respect to other, older or newer versions of the reference documents.
- The author used the English translations of the Ordinance on Political Rights (PoRO) and the Ordinance on Electronic Voting (OEV) as reference points. English is not one of the official languages of Switzerland for which these Ordinances will be legally binding even when finalized. Thus, errors and misunderstandings could result from translation issues.[1]

[1] The English translations of the PoRO and OEV point out: "English is not an official language of the Swiss Confederation. This translation is provided for information purposes only and has no legal force."

1.3 Key reference documents

The analysis in this report relies significantly on the key documents listed in Table 1, and was performed using the specific versions of the documents listed.

1.4 Overview of observations

Switzerland's E-voting program, and the SwissPost E-voting system in particular, have been evolving and under development for well over a decade. The current system now under audit represents a snapshot in time on that long-term evolutionary path. The current generation of the system under audit takes many important and valuable measures for security and transparency that are to this author's knowledge unprecedented or nearly-unprecedented among governmental E-voting programs worldwide. At a technical level, these measures include individual and universal verifiability mechanisms, trust-splitting of critical functions across four control components, the incorporation of an independent auditor role in the E-voting process, and the adoption of a reproducible build process for the E-voting software. At a broader process level, the opening of the system's specifications and source code to public review, the establishment of a bug bounty program for the system, and the regular involvement of independent, international experts in examining both the E-voting program as a whole and this E-voting system in particular, similarly represent unprecedented levels of transparency and rigor. The author sees ample evidence overall of both a system and a development process represent an exemplar that other governments worldwide should examine closely, learn from, and adopt similar state-of-the-art practices where appropriate.

Similarly as a snapshot in time along a long-term development path, the current system under audit is still far from the ideal system that the author of this report – or perhaps any expert well-versed in this technology domain – would in principle like to see. Some issues the author considers significant and worrisome are already well-known but currently placed explicitly outside the system's scope or threat model: for example, the current system's reliance on a trusted and fully-centralized printing authority, and its exclusion of coercion or vote-buying as a risk to be taken seriously and potentially mitigated. This report will not spend any significant time discussing these issues

Document name	Source	Version	Date
Ordinance on Political Rights (PoRO)	Swiss Confederation		28 April 2021
Federal Chancellery Ordinance on Electronic Voting (OEV)	Swiss Confederation		28 April 2021
SwissPost Voting System architecture document	SwissPost	v0.9.1	17 August 2021
Swiss Post Voting System specification	SwissPost	v0.9.7	15 October 2021
Protocol of the Swiss Post Voting System	SwissPost	0.9.11	15 October 2021
Operational Guide: E-Voting	SwissPost	V02.02	3 September 2021
Infrastructure whitepaper of the Swiss Post e-voting system	SwissPost		15 November 2021

Table 1: Key documents referenced in this report.

precisely because they are both well-known already and technically outside of the audit's scope.

In other respects, the current system incorporates important security and transparency principles that represent important steps forward, but which are not yet embodied as completely or systematically in the system's architecture, design, or implementation as they could be and ideally should be. This report will place much more emphasis on examining these areas, especially in Section 2 focusing on the system's overall architecture. In particular:

- Explicit documentation of the architecture's security principles and assumptions, and how the concrete system embodies them, is still incomplete or unclear in many respects (Section 2.1).
- The architecture's trust-splitting across four control components strengthens vote privacy, but does not currently strengthen either end-to-end election integrity or availability as would be ideal (Section 2.2).
- While the system currently takes many important measures to ensure the critical property of independence between control components, other desirable measures are not yet in place (Section 2.3).
- The architecture critically relies on an independent auditor for universal verifiability, but the measures taken to ensure the auditor's independence appear incomplete at least in documentation (Section 2.4).
- While the system's abstract cryptographic protocol is well-specified and rigorously formalized, the security of the lower-level message-based interactions between the critical devices – especially the interactions involving offline devices – do not yet appear to be fully specified or analyzed (Section 2.5).

While this report focuses primarily on architecture-level issues such as those above, it also identifies and discusses several issues the author identified that are largely specific to one of the three audit scopes: E-voting protocol (Section 3), software implementation (Section 4), and E-voting infrastructure (Section 5).

Some of the issues this report identifies – especially those of incomplete or unclear documentation – should be readily fixable at moderate cost in the near future. Some of the other highlighted issues, however, are more complex and involve difficult cost/benefit tradeoffs. Addressing some of these issues may involve significantly more development time and financial investment, and hence may be infeasible to implement in the current-generation system. That is, some of the additional measures this report recommends are most likely feasible only in another major generational redesign of the E-voting architecture and system, perhaps requiring another 5–10 years of continued investment and development. These observations are made again in the recognition that the current system represents only a snapshot in time within a long-term evolutionary path. The intent is to point out areas that could and should be improved not just in the current-generation system but in future E-voting program generations.

The author did not identify any critical security vulnerabilities in the current system that represent clear and uncontestable violations of the goals, assumptions, and threat model likewise laid out by the current-generation system. The author hopes that the major issues discussed by this report can and will be addressed substantially, sooner or later, at least before the E-voting system is deployed for unlimited use with the potential for becoming the predominant voting channel in Switzerland (as postal voting is predominant now). However, it is also clear that

the ideal E-voting system will in practice never be developed except through a long-term, continuous investment in improving, refining, evaluating, and periodically redesigning the system – and this required long-term investment is unlikely to be maintainable if the system is not usable and providing some value in the meantime.

Alternative voting technologies, such as the postal voting approach currently predominant in Switzerland, carry significant systemic risks of their own [3]. A complete and stagnant reliance on today's legacy voting systems could lead to a more-hurried adoption of less-mature and more-risky digital methods, down the road, when convenience-driven adoption pressures might become irresistible. Recognizing these complex realities, together with the fact that the current system – imperfect as it may be in many ways – embodies and often leads the state-of-the-art globally in the security and transparency measures taken, the author finds no clear evidence that the current system should not be used in a cautious, experimental, "safety-first" fashion, by a limited population of Swiss voters with the greatest need for the advantages of E-voting.

2 General architectural concerns

This section focuses broadly on the overall architecture of the SwissPost E-voting system. Beyond merely performing a "compliance check" between the SwissPost system design and the letter of the relevant draft regulations (the PoRO and OEV), this section raises concerns and discusses risks and tradeoffs more generally based on the author's understanding and judgment of the basic requirements and state-of-the-art practices applicable to E-voting systems in general.

2.1 Architecture documentation issues

The primary high-level architecture specification – titled "SwissPost Voting System architecture document" – is useful in understanding the "big picture" of the system but is currently missing substantial important information, and is not fully aligned with the other key documents.

Architecture overview: The document is missing at the start a clear description of the key components and roles - human, organizational, and electronic – comprising the system, what their respective basic purposes are, and how they relate to and interact with each other. A block diagram figure showing all the relevant entities and their relationships would be helpful. Section 3.1, titled "Business Context", includes a list of components, but mixes in discussions of requirements and threat model, which are also important but should be separated for clarity.

The current list is also incomplete, and not fully aligned with the components and roles described in the system's other key documents. For example, the list omits the Administration Board, mentioned briefly elsewhere but never defined. The list also omits the critical Setup Component that features prominently in the System Specification and Protocol documents, as well as the other critical canton-operated computers discussed in the Operational Guide (*e.g.,* Synchronization Computer and Decryption Computer).

Usage model: The architecture document could benefit from a clear high-level summary of how it is used: *i.e.,* the full "workflow" it supports including configuring an election, starting it, collecting votes, through closing and finalising an election. Further, the architecture document should summarize – or at least point to more information elsewhere – about what precise *kinds* of elections, ballots, and voter selections the architecture is (or is not) designed to support: *e.g.,* yes/no or multiple-choice questions, single-seat and multi-seat candidate elections, write-in candidate choices, etc. Is a voter required to answer all questions on a ballot, or can voters abstain or choose no answer to some questions? In multi-seat elections, can users choose the same candidate more than once? How are ballot formats configured in each of these cases? Lack of clear definition about how a critical system is and is not intended to be used can easily lead to subtle security risks, such as those discussed later in Section 3.1.

Security architecture: The architecture is missing a clear and well-marked description of the system's overall security architecture. For example, what are the precise security goals and security properties that the architecture should achieve? In what real-world context or under what conditions is the architecture expected to remain secure? Which roles and devices are required to be independently managed (*e.g.,* control components, auditors

and their verification computers), and in what ways is this independence expected to be assured operationally? As Section 2.5 discusses in more detail, the cryptographic security proofs in the Protocol specification cover only the high-level, *abstract* protocol and not the lower-level, concrete implementation in terms of messages or manual data transfers between devices. The system needs at least a clear summary of the system's "top-to-bottom" and "end-to-end" security architecture, not just a proof of the high-level abstract cryptographic protocol.

Threat model definition: While the document does discuss threat model in terms of which components are assumed to be "trustworthy" or "untrustworthy", this description of the system's assumptions is imprecise, not discussing for example what critical security properties the components are expected to satisfy – *i.e.,* what exactly are they trusted *for*, and what are they not trusted for? What classes of threats and adversarial capabilities is the system expected to be able to withstand, what classes of adversarial capabilities are considered out of scope, and what are the main justifications for considering this threat model suitable for E-voting in Switzerland?

Security analysis: Finally, although this document is not the appropriate place for detailed cryptographic proofs, it could nevertheless benefit from an informal but clear high-level security analysis section, describing from a "big picture" perspective how the system's components and their behavior fit together to make the overall system and end-to-end voting process secure. The security analysis should systematically address all the major identified threat vectors: *e.g.,* an attacker attempting to tamper with the election results, an attacker attempting to compromise voter privacy, an attacker attempting to take the system offline (denial-of-service attacks).

2.2 Limited application of trust splitting

Switzerland's E-voting program is unique in having taken the extremely important and positive step of mandating *trust splitting* as a basic part of the E-voting architecture. That is, some of the most security-critical processing operations are divided among multiple *control components* so that no single control component need be trusted completely. Given how security-critical voting is to any democracy in general, and given that we can never realistically expect any single device to be perfectly secure, the author feels that the lead Switzerland has taken in mandating trust splitting is one that should be followed by any other country deploying or considering deployment of an E-voting system.

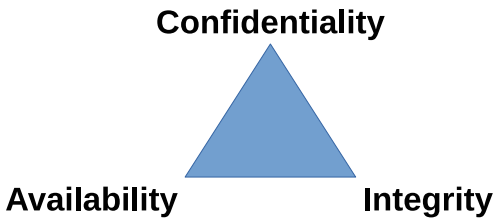

Figure 1: The AIC or CIA triad, representing the three main protection goals of classic information security: Availability, Integrity, and Confidentiality.

Trust splitting is a particularly powerful technique in that it can *exponentially* increase a system's trustworthiness, in a quantifiable and provable sense, at a moderate, only *multiplicative* cost increase. If an individual component has a probability p of encountering a relevant failure or security compromise in a given time period, for example, and we split trust across n fully-independent components such that the system fails only if all n components fail, then the probability of an overall system failure in the same time period is in principle exactly p^n. As a concrete example, if each individual component has a 1% failure probability in a year ($p = 10^{-2}$), then trust splitting across four such independent components in principle reduces the overall system's failure probability in a year to 0.000001% ($p^4 = 10^{-8}$). That is, at a moderate $4\times$ multiplicative cost in infrastructure, we obtain an exponential cost in effective trustworthiness. Many caveats apply, of course – actual failure probabilities are difficult to estimate, no set of redundant components are ever *fully* indpenedent, etc. – but the basic "exponential benefit at multiplcative cost" principle and opportunity remains.

Which security properties does trust splitting protect? There are many different ways in which trust splitting can actually be embodied in the architecture of a system, however, with different results and cost/benefit/risk tradeoffs.

A basic principle of information security is known as the AIC or CIA triad: Availability, Integrity, and Confidentiality, the three conceptually-orthogonal security properties that we generally want any information system to protect, as illustrated in Fig. 1. Trust splitting can drastically increase a system's protection in all, or only some, of these dimensions – but this depends on precisely how trust splitting is used in the architecture.

The SwissPost E-voting architecture document does not clearly specify which of the AIC properties the trust-splitting across control components is or is not intended to protect, nor does it justify the particular choices the architecture makes as to how trust splitting is used.

Trust splitting and Confidentiality: We can clearly infer from the system's design (and from the OEV's requirements) that trust splitting *is* intended to protect voter privacy – *i.e.,* the confidentiality of cast votes – beyond the baseline of a single fully-trusted component managed by one team. The fact that each of the four mixing control components (CCMs) – three managed by the SwissPost and the final one by the cantons – independently shuffles all of the ballots in turn should guarantee that voters remain anonymous even if all but one CCM is compromised. This property, again, represents an important step forward in public development and deployment of E-voting systems that is highly commendable. Provided that all the underlying assumptions and implementation details hold up (threat model and cryptographic assumptions, protocol and software correctness, etc.), the SwissPost E-voting system appears to succeed in providing this architecturally-enhanced level of privacy protection.

Trust splitting and Availability: The SwissPost E-voting system's control components are designed to operate in an n-of-n threshold or "anytrust" model [7], in which *all four* must participate and *none* can be offline in order for the system as a whole to remain available. From this fact we can infer that trust splitting is *not* intended to enhance availability in the SwissPost E-voting system beyond the availability of the underlying components. Protecting availability at architecture level would require a t-of-n threshold model where $t < n$, as is typical of [Byzantine] fault tolerant systems, for example [2].

This lack of availability protection in the architecture's use of trust splitting obviously represents a potential risk, especially since a system with four "anytrust" control components is in fact technically *less protected* against availability failures than would be a similar system with only one control component. (There are now four critical servers, in place of just one, whose downtime could interrupt E-voting in Switzerland.) This risk may well be justified, however, for example on the grounds that n-of-n "anytrust" systems are much simpler to design and build than t-of-n theshold systems, and that perhaps (all of) the underlying control components are expected to provide "good enough" overall availability on the basis of their individual high-availability provisioning and management by SwissPost. It would be preferable for the E-voting architecture document to discuss these issues explicitly, however, and to state clearly if – and why – the architectural trust splitting is not intended to enhance availability.

Trust splitting and Integrity: It is less clear whether and to what extent the architecture's trust splitting across control components is intended to protect the *integrity* of the voting process. The architecture document seems to suggest such an intent, for example in this sentence at the beginning of section 5.5.1:

> The *Control Components* work together as a group and participate in all the sensitive operations defined in the voting protocol to guarantee its integrity.

More generally, given how fundamentally critical the results of an election are to the transfer of power in a democratic country, we might hope and expect a state-of-the-art E-voting system to bring all available resources to bear in protecting election integrity. We might hope in particular that due to its trust splitting across four control components, the overall probability of election integrity failure due to a compromise in any one critical component should be reduced exponentially as in the ideal cost/benefit analysis sketched above – *e.g.,* reducing a 1% per-component failure probability to a 0.000001% overall integrity failure probability as in the earlier example.

The actual Swiss Post E-voting protocol, however, does not appear to be designed so that the control components systematically work together to protect the *integrity* of

the voting process, but only to protect its *privacy* as discussed above. For example, the mixing control components (CCMs) each *generate* a zero-knowledge proof of the correctness the shuffle they perform – but none of the control components ever actually *verify* any of the proofs generated by the other control components. Only the separate auditor(s) are described as verifying these proofs.

The OEV explicitly requires the system to have four control components, but leaves the number of auditors unspecified, apparently requiring only one. Further, the Operational Guide appears to be written under the expectation that there is only one Verification Computer, and does not mention the auditor role at all (Section 2.4). If an election indeed has only one auditor, then an attacker needs to compromise at most *two* critical devices in order to manipulate the election results undetectably: namely, any *one* control component, who replaces ballots arbitrarily and generates an invalid proof, plus the auditor's verifier device, which merely pretends to check it and claims that the the proof it "checked" was valid.

This weakness is probably not a compliance issue, because the OEV demands in Art. 8 only that the auditor-checked "proof" remain valid when all but one control component is compromised, and hence permits this critical dependence on the (perhaps one and only) auditor's trustworthiness.

Nevertheless, this weakness represents a significant "missed opportunity" in the system's security architecture. The main development and infrustructure *costs* of 4-way-redundant trust splitting have already been mandated and budgeted as part of the system, but the potential *benefits* that could be derived from those costs have been achieved only partially, for vote confidentiality but not for election integrity. The four control components already produce the necessary proofs anyway, and the code to verify the proofs has likewise already been written and must be maintained anyway for the auditor(s).

Recommendation: In a more robust mixing workflow, each of the control components would first check the shuffle proofs generated by the prior mixing components (CCMs) in the sequence before performing its own shuffle and generating its shuffle proof. Then, after the last mixing step, each control component would check *all* of the shuffle proofs in the sequence that it hasn't checked already, before "signing off" on the collectively-shuffled result. The signoffs of *all* control components should be required before any subsequent steps are taken, either manual or automated, such as the initiation of vote decryption. In this way, any single control component – *i.e.,* the unknown CCM assumed to be trustworthy – could protect integrity throughout the process as robustly and redundantly as the CCMs now protect confidentiality, independent of how many auditors there are or their actions.

More generally, proactively checking the correctness of "everything that was done so far" in the control components before starting to perform "the next" operation is just good defensive security engineering practice, and may help head off future potential vulnerabilities similar to the recently-discovered "decryption oracle" issue #YWH-PGM2323-35.

2.3 Control component independence

Even if the architecture relies on trust splitting across the control components to protect only confidentiality and not integrity or availability, as discussed above, it is still important – and explicitly required by the OEV in Art. 8 – that the control components be as *independent* as possible. Independence ensures that an attacker who somehow manages to compromise one control component hopefully has a small probability of being able to exploit the same weakness to compromise the other control components.

In principle, if each control component is fully-independent and has a probability p of privacy failure, then the overall system has the exponentially-smaller probability p^4 of privacy failure, as discussed above in Section 2.2. At the other extreme, if the control components are fully *interdependent* – that is, not at all independent of each other – then a successful (privacy) exploit against one immediately leads to a successful (privacy) exploit against the others, so the overall system fails with exactly the same probability p as the individual failure of any single control component, yielding no benefit.

Documentation considerations: In practice, of course, redundant systems of this kind lie somewhere between these two extremes, achieving some level of independence but never perfect independence. It is therefore important that the architecture explicitly address and document the

ways in which the control components are – and are not – expected to be independent in actual operation, and to justify these choices. The architecture document currently contains no discussion of these independence considerations, which should be added.

For example, the fact that the architecture incorporates four (partly-)independent control components at all, and are run on separate "bare metal" machines, represents a significant step forward in the state-of-the-art for deployed E-voting systems. The system already takes the additional important steps of ensuring that these control components are managed by separate teams, that they run on a diversity of hardware (*e.g.,* Intel and AMD processors), that they use a diversity of operating systems (*e.g.,* Windows and Linux), and that they are physically isolated in separately-locked cages. These general independence provisions should be clearly described as part of the publicly-documented architecture, even if not all details of precisely *how* these provisions are implemented need (or should) necessarily be made public.

Just as importantly, the architecture should disclose, analyze, and ideally justify the ways in which it cannot, or currently does not, assure independence between the control components. Some of the potential independence limitations apparent in at least the current version of the system include in particular the following:

Software interdependence: There is currently only one *software implementation* of the control component logic. If a security-critical bug in one control component goes uncaught by the SwissPost's bug bounty program, therefore, then that same bug may likely exist and be exploitable in all the other control components.

N-version programming – in which independent development teams each write separate implementations of the control component logic while referring only to a common functional specification – could mitigate this risk, though obviously at a high development cost.

Platform interdependence: Even if the control components run diverse operating systems, many of the libraries, runtimes, and other platform elements that the control components depend on may be cross-platform. Thus, one critical security bug in a single cross-platform library that all four control components happen to depend on might render all four control components simultaneously vulnerable. The fact that the control component software is largely written in the portability-focused Java language, while clearly a "feature" for development and maintenance purposes, may actually be a "bug" from an independence perspective, because so much cross-platform Java code may in fact be identical in practice across the four control components. The recent "log4j" vulnerability, which simultaneously affected a large fraction of the entire Java ecosystem, is a prominent case in point.

A first step in addressing software dependency risks of this kind would be to perform (and maintain) a *deep dependency analysis* of the software running on the control components [8], to identify software components potentially buried "deep in the stack" that might unexpectedly be identical (and hence perhaps identically exploitable) across most or all of the control components. In cases where there are multiple alternative implementations of these deep dependencies – as there often are for popular open-source software systems – the control components could and ideally should be configured to rely on diverse alternative implementations of these components. Performing and maintaining such a deep dependency analysis would incur nontrivial development costs, of course, and many of the common dependencies it identifies might *not* have readily-available and "pluggable" alternatives. In the long term, therefore, new alternative versions of these dependencies might have to be developed "from scratch" as in N-version programming to achieve the ideal of full software platform independence, a goal that may be conceivably feasible in the (very) long term but might well be far too costly to be financed by any one country's E-voting program alone.

Organizational interdependence: Three of the four mixing control components (CCMs), and all four of the return code control components (CCRs), are currently deployed and managed by the same organization (SwissPost) and physically hosted in the same data center infrastructure (SwissPost's), even if under the management of different teams. These organizational and hosting dependencies present common-mode failure risks at infrastructure level – one which is discussed more specifically later in Section 5.2.

These dependencies could in principle be mitigated by contracting with other organizations independent of SwissPost to host and manage some of the control components, obviously at certain substantial costs both financial and logistical.

Persistent state interdependence: The system architecture document states that "Both the control components and voting server persist to a standard RDBMS (Oracle)". This appears to be a *common* RDBMS for all four control components, and thus represents a significant potential independence risk that the architecture should clearly document and analyze.

In particular, the four control components each have internal state that must be persistent (*i.e.,* survive reboots) but whose (independent) confidentiality is crucial – such as the CCR and CCM private keys that the protocol uses. Do these critical secrets persist to the common RDBMS as well, or are they stored only locally on the physical control components? If an attacker could learn the cryptographic private keys of all four control components merely by compromising the common backend database that they persist to, then this would represent an independence failure of high concern.

In general, is the persisted control component state encrypted? If so, by whom (*i.e.,* by the control component itself or by the RDBMS server), how are the keys to this encryption managed, who holds them, and how are they recorvered after a failure? There are reasonable and probably-secure answers to all of these questions, but the architecture needs to address them clearly.

Infrastructure interdependence: The control components' dependence on a common RDBMS for persistence is just one example of a common infrastructure dependency, and any such common infrastructure dependency should be clearly documented, analyzed, and justified on a basis of costs, benefits, and risks.

Another such common infrastructure interdependency of potential concern is the control components' reliance on a common Splunk instance to gather and archive the logs of all control components. An adversary who could compromise this common Splunk instance might effectively be able to cover traces of attempted (or perhaps successful) attacks against all four control components at once. Even if the control components also maintain independent local copies of their logs, this might be of little benefit if the SwissPost teams managing the control components routinely only monitor the logs via the common Splunk instance.

Still another infrasructure dependency of potential concern is the use of a common internal SwissPost "dashboard" service for the general control and monitoring of the control components. If an adversary were to compromise this common dashboard service, then it might successfully convince all four teams that "all is well" with their respective control components, potentially for an extended time period, when in fact all is decidedly not well with one or more of them. This interdependency risk exemplifies the classic "trusted path" security principle: for a human (person or team) to access a machine (*e.g.,* control component) securely and independently, the complete access path between the human (team) and machine (control component) must be secure (and independent).

A common potential interdependency risk worth noting is that many of the common infrastructure dependencies discussed above are in fact *virtualized services* hosted atop a common SwissPost infrastructure cluster and managed by a common virtualization platform (VMware). This dependence on virtualized services thus presents the further risk that a successful exploit against the underlying virtualization platform – *e.g.,* a VMware escape or management system exploit – could effectively compromise *all* of these infrastructure dependencies at once (*e.g.,* the RDBMS state, the Splunk logs, *and* the dashboard services for *all* the control components). These infrastructure dependency risks, both direct and indirect, need to be clearly documented, analyzed, and ultimately justified.

Supply-chain interdependence: While the control components are managed by separate SwissPost teams once they are procured and installed, the actual process of provisioning them currently follows common, centralized internal processes for procuring and configuring new hardware. This common process could potentially expose all the control components together to a variety of "supply chain attack" threats.

Suppose, for example, an adversary successfully compromises (either at human or electronic level) the SwissPost's internal hardware procuration pipeline, or the pro-

cesses of a hardware supplier or other supply-chain intermediary that SwissPost usually relies on. The adversary might then realistically be able to intercept or divert deliveries of all four of the control components – in order to compromise them at boot, firmware, or hardware level – even if the four control components are from different hardware vendors and ordered at different times. This form of supply-chain attack is a known practice of nation-state intelligence agencies, for example [5].

To maximize supply-chain independence, it would be preferable if each of the independent *teams* tasked with managing each control component were also responsible for procuring the hardware – through independent purchasing channels distinct from the SwissPost's normal internal process for at least some of the control components, and preferably while minimizing any advance leakage of any information about when, through what channels, or for what purpose the hardware is being purchased. Similarly, for maximum independence it would be preferable if *only* the members of each control component's responsible team ever had physical access to the hardware, from the moment of purchase and delivery through installation in the locked cage dedicated to that control component, and thereafter any time physical access is required. These independence considerations must be balanced against cost and other practicality considerations, of course.

2.4 Auditor independence

Particularly because the SwissPost E-voting architecture relies almost *solely* on the role of auditor(s) to verify the integrity of critical E-voting processes such as mixing and decryption, as discussed above in Section 2.2, it is particularly crucial that the implementation of the auditors' role be as independent as possible from the implementation of the rest of the system.

Just as with the independence of the control components, the architecture should clearly document the provisions taken to ensure the auditors' independence both organizationally and technically, and similarly the limitations of this independence and the possible steps *not* taken with relevant justifications. For example, how many auditors are there expected to be and how are they selected? How are they expected to procure and manage their critical devices – termed somewhat inconsistency the "auditors' technical aids" in the OEV, the "verifiers" in the System Specification and Protocol, and the "Verification Computer" in the operational guide?

One particular area of concern is that while the Architecture document, the System Specification, and the Protocol describe the auditors and their technical aids or verifier devices as clearly separate and independent roles, the current Operational Guide never mentions auditors at all as an independent role, and stylistically documents the "Verification Computer" as being merely one of "four computers (in the form of notebooks)" that are used to manage the election. In general, the Operational Guide appears to be written as if for an audience consisting of *one person or team* who manages all of these security-critical computers together including the (auditor's) Verification Computer. The Guide makes no clear or explicit separation between the steps that are supposed to be performed by the team responsible for running the the election itself (via the Configuration, Decryption, and Synchronization Computers), and the steps that are supposed to be performed by the independent person or team filling the auditor role and thus presumably responsible for the Verification Computer. While this is most likely just a readily-fixable documentation issue, it is nevertheless architecturally important for the auditor role to remain separate and strongly independent from the rest of the system not just "in theory" (*e.g.,* in the cryptographic protocols and proofs) but also in the everyday operational practice that the Operational Guide outlines.

2.5 Message-level authentication and integrity protection design

The SwissPost E-voting system represents a fairly complex protocol involving numerous interactions by numerous devices and subsystems: the voter' devices, the voting server, the control components, the cantonal computers, the verification computer, etc. For a concrete implementation of such complex protocol to remain secure, it is generally critical to ensure that the *interactions* between all the relevant electronic components are properly secured: *e.g.,* to ensure that messages that the protocol defines as coming from a particular device *can only* come from that particular device and can be verified as such (authentication), and that when a second device receives and consumes information produced by the first device, that the

information consumed by the latter is indeed identical to the information supposedly produced by the former (message integrity protection).

2.5.1 The benefits, and risks, of abstraction

Neither the System Specification nor the Protocol document thoroughly address this issue of securing the interactions between most of the devices, because these documents are written (and formally analyzed) at a high level of abstraction representing an abstract (ideal) rather than a concrete (implemented) protocol. This high level of definition and analysis corresponds to common practice in the cryptographic theory community, and hence is not a problem *per se*: defining and analyzing protocols at a high abstract level makes the analysis problem more modular and (relatively) tractable, whereas it would often be intractable if all of the concrete lower-level details of the interactions between devices (*i.e.,* precisely how each device sends and receives each message) were included in the analysis as well.

For example, in interaction diagrams like Fig. 6 "Overview of the SetupVoting algorithm" in the System Specification, and the corresponding text, device interactions are specified only in terms of abstract messages. For example, each of the CCRs send a message containing $\mathbf{pk}_{\mathrm{CCR}_j}$ to the Voting Server, the Voting Server sends a message containing $\{\mathbf{pk}_{\mathrm{CCR}_j}\}_{j=1}^{4}$ to the Setup Component, and so on. At this level of abstraction, the System Specification and Protocol merely *assume* – usually implicitly – that the Voting Server somehow knows that the first message indeed *must have* and *could only have* come from the correct CCR 1 through 4 and not from an attacker trying to impersonate one of the CCRs (authentication), and that the exact message that each CCR sent is received correctly and unmodified by the Voting Server (message integrity protection). Again, it is common practice and generally a good idea in a high-level system specification or cryptographic protocol to abstract away from these low-level details of interaction security – provided at *some* lower level these implementation details of interaction security are handled carefully and are appropriately documented elsewhere.

For *online, networked* interactions between devices it is fairly common merely to assume that a standard lower-level protocol such as Transport Level Security (TLS) handles the authentication and message integrity protection of the point-to-point interactions between devices. The SwissPost E-voting system documents appear to assume implicitly that such a layer exists and is successfully securing device interactions, but the documents never *explicitly* describe these assumptions about lower implementation layers, even briefly. For example, neither TLS nor message integrity protection appear to be mentioned in any of the documents. The system architecture document includes the relevant terms "PKI" (Public Key Infrastructure) and "MAC" (Message Authentication Code) in its list of terms, but never actually refers to them elsewhere or mentions how they are used or relevant in the system's security architecture. The system architecture document or a related document should at least summarize the principles and methods by which the interactions between all the relevant devices are secured at the message interaction or or data transfer level.

Merely assuming (implicitly) that a layer like TLS is present and "silently doing its job" is problematic in the Swiss Post E-voting protocol in at least three ways:

Threat model: First, some of the devices are assumed in the threat model to be potentially-compromised (including the voters' devices and the voting server), but in practical deployment it is still extremely important that these interactions be properly authenticated and integrity-protected. As a straw-man example, while *not* securing any of the connections between voting devices and the Voting Server via TLS (*i.e.,* just using "legacy" unencrypted HTTP) might technically not be a violation of the abstract threat model since *all* of these devices might technically be compromised, nevertheless in practice the lack of such protections would immediately expose the entire voting system to trivial and devastating denial-of-service (DoS) attack opportunities at the very least.

Public key infrastructure: A second way in which just "assuming TLS" is inadequate is because TLS-based authentication in turn fundamentally assumes and depends on centralized Public Key Infrastructure (PKI). In particular, TLS's assumed PKI model assumes that all of a number of global root *certificate authorities* (CAs) – and often many of the subsidiary CAs that they delegate signing power to – are fully trusted and never (for example)

issue bad certificates allowing an attacker to impersonate a given domain name or other identity.

If the SwissPost E-voting system depended fully on TLS's traditional centralized PKI model for authenticating the interacting devices throughout the E-voting protocol, then this would be a major security concern, especially in terms of the independence considerations discussed in earlier sections. If all the system's CCs, and/or the cantonal and verification computers, authenticated each others' messages via traditional PKI certificates issued by any of the standard global root CAs, for example, then an attacker would need to compromise only a *single* root CA in order to impersonate *any and all* of the critical devices (CCs, cantonal devices, verification computer, etc). Worse, the attacker might perhaps need to compromise only a single *subsidiary* CA that has sufficient delegated signing authority over the relevant portion of the domain namespace – *e.g.,*, over '`.ch`' domains perhaps.

Fortunately, based on the author's discussions with SwissPost, it is clear that the E-voting system does *not* rely naively in this way on TLS's traditional centralized PKI. However, it is a problem that none of the current documentation adequately describes how cross-device authentication or PKI actually works in the SwissPost E-voting system and on what basis these message interactions should be deemed secure.

Manual data transfers: Finally, a third reason that just implicitly "assuming TLS" is inadequate is because a number of the most critical cross-device interactions in the SwissPost E-voting system are (for good reasons) *not* online or network-based, but instead via physical *data carriers* (*i.e.,* USB sticks) used to transfer information manually from one device to another when offline ("air-gapped") devices are involved. For such manual information transfers, TLS is obviously not in use and is unable to provide either message authentication or integrity protection for these transfers. We might hope or expect that the mere fact that these transfers are done in the presence of various witnesses (*e.g.,* Electoral Authority and Administration Board members) to confer adequate security to these information transfer events. But such a hope is not adequately well-founded, as explained further below, in that a single compromised device might for example try to trick the in-person witnesses (and perhaps succeed).

The current Operational Guide briefly refers in places to signing and signature checking (e.g., Step 4 on page 12), but neither this nor relevant documents appear to state where these signing keys come from, how they are issued and managed, what the trust assumptions are around the management of these keys, or whether and how signing or other integrity protection comes into play in the many offline cross-device interactions *other than* the few for which these explicit signing processes are mentioned.

Thus, the system's message-level interaction and design – particularly its methods of authentication and message integrity protection throughout the protocol – are worrisomely incomplete at least from a documentation perspective. Either the system architecture document, or some other appropriate document, should clearly define the principles by which not only the *online* (networked) but also the *offline* (data carrier) interactions are authenticated and integrity-protected throughout the protocol. If messages or information on data carriers is signed, then the architecture at least needs to summarize what form of PKI it assumes, how the signing keys are created and managed, and so on. More broadly, not just the abstract protocol but its concrete implementation needs to ensure that there are appropriate integrity checks not just for particular pairwise device interactions but also "end-to-end" when we consider the participating humans (*e.g.,* Electoral Authority and Administration Board members).

2.5.2 Decrypted results versus verified results

To place the above interaction-security issues into perspective, we now outline a hypothetical attack that appears *potentially* feasible when judging only by the current documentation. Whether or not such an attack actually *is* feasible in reality depends on further implementation and operational details the author has not completely analyzed, so this concern does not represent a claim of an actual, exploitable security vulnerability.

As discussed earlier in Section 2.2, for election integrity the current system's design critically assumes that either *all* of the mixing control components (CCMs), or the verification computer, is uncompromisd. Let us therefore assume that the verification computer is uncompromised, but suppose that the Decryption Computer, which performs the final mixing and decryption of election results, is compromised. Can the Decryption Computer po-

tentially manipulate the election results and display the manipulated results to the witnesses physically present, while sending the Verification Computer a *correct* set of proofs and election results, without the trickery being detected? In particular, can the Decryption Computer potentially "decrypt" and "output" a different set of election results from those that the Verification Computer has (correctly) verified? Nothing in the either system architecture document, nor the system specification or protocol, nor the operational guide, currently seems to indicate that there is a concrete correspondence check between the results the Decryption Computer outputs and the results the Verification Computer has actually verified.

This hypothetical attack might proceed as follows:

1. The first three online mixing control components (CCMs) correctly shuffle all the ballots and produce correct shuffle proofs.

2. The Decryption Computer receives the correct encrypted ballots from the prior CCMs via Data Carrier 9 (Operational Guide section 7.3, Step 1).

3. The compromised Decryption Computer performs a *correct* shuffle and produces a *correct* shuffle proof, and exports this correct information to the Verification Computer via the Data Carrier Verifier (section 7.3, Step 2).

4. The Verification Computer checks the (correct) final shuffle proofs and reports "Success!" to all the witnesses present The Verification Computer produces its PDF verification report, which the present members of the electoral authority sign as required.

5. During subsequent decryption of the election event (section 7.5), however, the compromised Decryption Computer decrypts the correct election results (*e.g.,*, "Alice 15, Bob 9"), but then *displays* a different, manipulated set of results to the witnesses present (*e.g.,*, "Alice 12, Bob 14").

6. The compromised Decryption Computer again forwards the *correct* results to the Verification Computer in the Step 5 export "for Final".

7. The Verification Computer receives this information, again verifies it successfully (because it is for the correct results), and again outputs "Success!" The Electoral Authority members see no sign of a problem and hence sign the second PDF verification report.

This form of attack may not be – and hopefully is not – feasible if (1) the Verification Computer's output in the final step includes all information required to perform a full "end-to-end" check of the *exact correspondence* between the results the Verification Computer verified and the results the Decryption Computer announced, and (2) the Electoral Authority members physically present always carefully check that correspondence. The correspondence information (1) might for example consist of a secure hash (*e.g.,* SHA256) of all the election results, and the correspondence check (2) might consist of explicit instructions that the Electoral Authority members each manually check that the hash displayed by the Decryption Computer and output in the announced results is identical to the hash displayed by the Verification Computer, indicating *which* election results were successfully verified.

However, none of the current documentation explicitly indicates either (1) that the Verification Computer even produces and displays the information required to perform this correspondence check, or (2) specifies that the witnesses present (*e.g.,* Electoral Authority members) are actually expected and required to perform such a correspondence check. Given that the lack of such a correspondence check would lead to a devastating election-integrity weakness in the presence of only one compromised device (the Decryption Computer), it seems quite important to document explicitly and analyze carefully the mechanisms and processes in place to prevent such attacks.

This concern, which again does not necessarily represent an exploitable vulnerability, is merely intended to illustrate one of a large class of risks that can arise if the messages or information exchanges between devices are merely assumed implicitly to follow a high-level abstract protocol without adequate, explicit and well-documented authentication and integrity checking at the lower message-based device-interaction level as well. This attention to lower-level interaction details is especially important in the SwissPost E-voting system context because the easy and typical, implicit "assume TLS" assumption is not directly applicable. As discussed above, TLS's threat model does not correspond to that of the E-voting system, TLS assumes traditional centralized PKI

unsuitable for the E-voting system, and TLS's protections could at best cover only the online interactions, and not the "data carrier" interactions involving offline devices.

3 E-voting protocol (Scope 1)

This section focuses on observations and concerns about the SwissPost E-voting protocol, as part of Scope 1 of the audit. The author did not attempt to perform a detailed review of the complete cryptographic protocol or its proofs, but inspected these aspects of the system as needed as part of a high-level architecture analysis of the system.

3.1 Usage guidance on ballot configuration

The E-voting system'ss architecture document currently leaves the intended usage model largely unspecified (Section 2.1), and this underspecification of usage model unfortunately carries into the current System Specification and Protocol documents as well. In general, neither document clearly specifies what kinds of questions and elections are and aren't safely supported by the system, and how verification card sets and ballots should be configured accordingly.

More specifically, the current System Specification and Protocol both leave unanswered the following questions, whose answers appear to be important to the protocol's security:

1. Can a voter cast a valid ballot that leaves some allowed selections unanswered?

2. In a multi-seat election allowing the user to make $k > 1$ selections, can a voter make fewer than k selections on a valid ballot?

3. In a multi-seat election allowing the user to make $k > 1$ selections, can a voter choose the same candidate more than once?

4. How should voting card sets be configured to handle the above cases?

5. How should the voting client formulate cast ballots in the above cases?

Optional selections: It is common in many voting contexts for choices to be optional: *i.e.,*, the voter is allowed but not required to answer all questions or to pick candidates in all elections. Neither the System Specification nor the Protocol describe how such optional choices should be handled in either verification card set (VCS) configuration or ballot casting.

As currently written, in fact, the System Specification and Protocol implicitly appear to suggest a possible approach that proves to be dangerously insecure. In particular, one of the key configuration parameters for a voting card set, ψ, is defined as "Allowed number of selections" (System Specification, "Symbols", page 6). The use of the word "Allowed" here suggests that a valid ballot might contain *up to* ψ selections but might legitimately contain fewer selections. If all of these selections were considered mandatory, in contrast, we might expect ψ to be defined as "Required number of selections".

The writing elsewhere similarly suggests that these ψ selections may be optional. When ψ is first discussed in section 3.2 "Election Event Context" of the System Specification, for example, it is introduced as "the number of selectable voting options ψ" (and not, for example, "the number of voting options ψ the voter must select"). Section 3.3.1 "Voting Options" starts with the statement, "A voter can select ψ out of n voting options", which again suggests that a voter need not make all ψ selections. If this were not the case then we might expect the sentence to read, for example, "A voter selects ψ out of n voting options" – or even more clearly, "A voter is required to select ψ out of n voting options".

If these selections are indeed optional as the writing suggests, then the next question is how a voting client is expected to formulate and cast a ballot in which some of the ψ allowed selections for a voting card set are missing: that is, how should a correct voting client use Algorithm 5.2 `CreateVote` to handle this case?

One potential (and dangerously wrong) interpretation might be that the value of ψ passed to `CreateVote` in this case would be the number of *actually selected* options, which might be less than the value of ψ that the voting card set was configured for. In this case, we might expect that `CreateVote` simply creates a "short" (but ostensibly still-valid) ballot.

A second (also dangerously wrong) supposition might be that the voting client is expected to "pad" the user's

selections to the standard number ψ for the relevant voting card set before invoking `CreateVote` – for example, by simply duplicating the last choice the user actually makes.

Both of these conceivable answers would be dangerously wrong, however, because using the protocol in this way would break both voter privacy and the cast-as-intended verifiability property as we explore next.

Privacy hazards: A "short ballot", containing fewer than the allowed ψ number of selections for the relevant voting card set, would be clearly distinguishable (by the untrustworthy voting server for example) from other ballots containing exactly ψ selections, which would decrease voter anonymity.

A "padded ballot" containing duplicate copies of the same option in more than one of the ψ selection positions, similarly, would be clearly distinguishable in cleartext from ballots not padded this way (or from ballots with a different number of padding selections).

Cast-as-intended verifiability hazards: If a "short ballot" with fewer than the standard ψ selections was valid, then a compromised (*e.g.,* malware-infected) voting client could undetectably "fill out" any or all of the voter's unfilled selections in any fashion of the adversary's choosing. The malicious voting client would then get back from the voting server *more* than the number of short voting codes it "needs" according to the user's actual selections, and would merely drop the short codes for the secretly-cast options and show the user only the (correct) short codes for the user's selections.

Similarly, if a valid ballot might be "padded" with duplicate copies of the same option, then a compromised voting client could silently replace any of these duplicate copies with non-duplicate choices before submitting the ballot to the voting server. The malicious client will again get back all the short codes it "needs" to show the user, plus additional codes for its secretly-cast (no-longer-duplicate) choices, which the malicious client simply discards instead of showing to the user.

Multi-seat elections: Section 3.4 "Correct Combination of Voting Options" in the System Specification, as well as the corresponding section 10.4 in the Protocol, imply that it is valid for some elections to allow the user to make more than one (*e.g.,*, $k > 1$) selections among the same set of options (*i.e.,*, corresponding to the same correctnessID). As with the question about the handling of optional selections discussed above, this multi-selection case raises the some issues above in the event the user can (and does) choose *fewer* than the allowed k choices in a particular ballot. If choosing fewer than k selections were to produce either a "short ballot" or a "padded ballot", then both privacy and cast-as-intended verifiability would be compromised as discussed above.

Further, the multi-seat election scenario presents the additional situation in which a voter might be legitimately allowed to choose *the same candidate* more than once in a single ballot. That is, if I am allowed three selections in a particular multi-seat race (such as the "correctnessID = ccccc3" mentioned in the specification), can I choose the same candidate, *e.g.,*, Alice, in all three selections, in order to cast all of my three votes to support Alice and none to support other candidates? This semantically-meaningful duplicate selection of a single candidate is in fact allowed in some elections in Switzerland, so this is not a purely-hypothetical corner-case. If a voting client were to handle this situation in the "natural and obvious way", however – by simply including the same option (and corresponding prime number) representing Alice in more than one of the ψ selections comprising the ballot, then both privacy and cast-as-intended verifiability would again be broken as discussed above.

For a multi-selection election in which up to $1 < d \leq k$ duplicate choices are allowed, it appears that the only safe way to use the system is to ensure that the user has d *distinct* options representing the same candidate to choose from. Each of these distinct options maps to a distinct prime number, and each results in a distinct code being returned if selected.

Furthermore, this *uniqueness-of-selections* requirement must be rigorously enforced by the voting client. If a voter could accidentally choose the same "Alice" option twice (instead of choosing the two distinct "Alice" options) when casting two votes for Alice, then the voter's ballot would be both distinguishable, violating voter privacy, and would be malleable by a malicious voting client, violating cast-as-intended verifiability.

Recommendation: First, the System Specification and Protocol should be updated to specify precisely what kinds of questions and elections are and aren't allowed by the system. The specifications should in particular describe how both optional choices and duplicate choices are to be properly handled, in voting card set configuration and in ballot formation.

The System Specification and Protocol documents should be revised to clarify that the ψ configuration parameter represents the number of selections the user is *required* to make, and not merely the number the user is "allowed" to make, on a valid ballot. If a choice is semantically optional for a user, then that must be handled by including an explicit pseudo-choice, such as "Abstain" or "No Answer", on the ballot, mapping to a unique prime number and producing a unique code for cast-as-intended verification.

The System Specification and Protocol documents should be revised to clarify that at ballot formation and casting time, all of the ψ required selections must be *unique* at protocol level – *i.e.,* mapping to *distinct* prime numbers that will yield distinct codes for cast-as-intended verification. If a voter is legitimately allowed to make up to d duplicate choices of the same candidate in a (multi-seat) election, then the documentation should clarify that the voter must be given d distinct options associated with that candidate, each mapping to distinct prime numbers and producing distinct verification codes if selected.

Finally, the System Specification and Protocol should be updated so that the uniqueness of the ψ selected options within a ballot is *enforced*, both by the voting client (*e.g.,* in the specification of Algorithm 5.2 `CreateVote`), *and* in the trustworthy part of the system (*e.g.,* perhaps in the specification of Algorithm 5.1.3 $\texttt{VerifyBallotCCR}_j$). If the uniqueness requirement is not enforced by the voting client, then a user might accidentally attempt to cast an invalid, privacy-compromising ballot, as described above. If the uniqueness requirement is enforced *only* by the voting client and not by the trustworthy components of the system, then a compromised (*e.g.,* malware-infected) voting client might secretly cast "invalid" ballots containing duplicate options on behalf of the user, which could then go undetected throughout ballot casting and mixing, but would effectively "mark" the voter's ballot, making it clearly distinguishable from other ballots throughout the process.

3.2 Write-ins and individual verifiability

The System Specification and Protocol include a capability allowing the voter to cast *write-in* options, in which the voter provides an arbitrary string that gets encrypted and included in the ballot separately from the predefined choices. However, this capability is incompletely documented. Furthermore, it is not clear whether this capability may be used at all in compliance with the applicable draft regulations (OEV), without modifications to those regulations.

Incomplete documentation: As with the issues of optional or duplicate choices discussed above, the System Specification and Protocol are lacking any clear description of precisely *how* the write-in capability is intended to be used, or can be used securely.

For example, voting card sets include a configuration parameter $\hat{\delta}$ such that $\hat{\delta} - 1$ is the number of write-ins permitted. This $\hat{\delta}$ in effect determines the total number of ElGamal ciphertexts comprising each ballot: one ciphertext containing all the predetermined choices and an additional ciphertext for each potential write-in (see for example Algorithm 6.2 $\texttt{MixDecOnline}_j$ in the System Specification).

However, neither the System Specification nor Protocol describe exactly *how* any write-in choices are to be encoded and encrypted to these $\hat{\delta} - 1$ additional ciphertexts. Neither does the documentation specify how these write-in choices *interact with* the predefined choices encoded into the main (first) ciphertext.

For example, if the voter makes a write-in choice for some election in which this is allowed, then that voter clearly must make one fewer regular choices of predefined candidates as part of the ballot's main ciphertext. What rules apply to this process, and how and where are they enforced? Must an election allowing write-in candidates define one or more predefined "placeholder" choices – such as "Write-in 1", "Write-in 2", etc., in case multiple write-ins are allowed in a particular election?

If write-ins are allowed in multiple *distinct* elections with different correctnessID values, then what is the mapping between selections encoded into the main ciphertext with these correctnessID values and the $\hat{\delta} - 1$ write-in ciphertexts? How are ballots potentially containing write-ins validated and counted after mixing and decryption?

Implementations of the protocol that answer these questions in the "wrong" way could readily lead to critical security bugs, such as eliminating cast-as-intended verifiability for *all* voters including those who make no write-in selections, as discussed below.

Regulatory compliance: Beyond the easily-fixable issue of incomplete documentation, a second important, and perhaps harder-to-fix, question is whether *using* the system's write-in capability is, or even *can be*, compliant with the regulations as defined in the draft OEV.

Write-in options cannot in practice provide "cast-as-intended" voter verifiability using any code-based system of the type the SwissPost system uses. If I intend to cast a write-in vote for "Mickey Mouse", then malware on my voting client can trivially change my ballot to a write-in vote for "Donald Duck" without my knowledge.

To guarantee cast-as-intended verifiability for write-in votes, each voter would have to be supplied in advance with codes not only for the predefined choices, but also individual codes for *all possible* write-in candidate names. Since this number of codes would be exponential in the number of characters allowed for write-ins, each voter would have to be mailed not just a voting card in advance, but a hefty printed volume, even if write-ins were constrained to be quite short. This is clearly not practical.

Section 13.3 "Write-ins" in the Protocol specification already acknowledges this problem: "the protocol cannot provide sent-as-intended for the write-in candidates, since it is impossible to map all possible write-in values to a Choice Return Code." However, the immediately-prior sentence of this paragraph also claims that "the existence of write-in candidates does not impact the security of the protocol" – but this claim could fail to be true depending on how the protocol is used, as discussed below.

Even if the write-in facility can be and is used so as not to affect the security of voters who make only predefined choices, it is not clear that at least the current draft regulations in the OEV even *allow* exceptions to the cast-as-intended verifiability requirements for voters who make write-in choices. Specifically, Art. 5 of the draft OEV requires that "The person voting can ascertain whether his or her vote has been manipulated or intercepted on the user device or during transmission" and that "the person voting receives proof that the trustworthy part of the system (Art. 8) has registered the vote as it was entered by the person voting on the user device as being in conformity with the system." By the most straightforward interpretation, this text would appear intended to apply to *all* voters who cast their votes electronically. No exception or "loophole" is readily apparent that would suggest that it is allowable and compliant for *some* electronically-cast ballots – such as those cast with write-in options – not to guarantee the individual verifiability property.

This gap between what the draft regulations seems to demand, and what the SwissPost system – or *any* similar code-based voting system – can actually deliver in the case of write-in options, needs to be resolved somehow. The obvious choices are:

1. Change the OEV to allow an exception to the individual verifiability requirement in the case of write-ins.

2. Disallow the use the write-in capability for electronic ballots in Switzerland governed by the OEV. That is, simply require that $\hat{\delta}$ always be configured to be equal to 1, at least in Switzerland. Voters could be instructed that if they wish to cast a ballot that includes any write-in candidates, then they must use a different voting channel (mail or in-person).

3. Simply remove the write-in capability from the voting system entirely, with the benefit of simplifying it slightly, if it is determined that its use is not and cannot be made compliant with the applicable regulations.

Election manipulation hazards: In considering how to resolve this issue, it should be recognized that while write-ins are *usually* rare in *most* elections, this need not be and is not the case in *all* elections. For example, there can be rare cases in which a "dark horse" candidate joins a race late, or is otherwise ineligible to be included in the predefined choices, but nevertheless proves extremely popular – occasionally perhaps even electable.

In such rare scenarios, in which many (perhaps even a majority of) voters cast write-in votes – however uncommon such scenarios might be – is it acceptable to lose individual verifiability for all of those voters?

We might, for example, envision a nation-state adversary "laying in wait" for exactly such a scenario – or

even deliberately trying to *manufacture* such a scenario, by throwing substantial resources at promoting some, perhaps *any*, "dark horse" candidate late in an election. The adversary need not even care who this potentially-electable dark horse candidate actually is or what their platform would be. But if this scenario actually occurs, and a majority of voters intend to choose "Alice" as their write-in candidate, adversary-controlled malware on the voting clients might secretly change most or all of those write-in choices to "Eve", the candidate the adversary wants to win. So while Alice is the dark horse candidate that became popular at the last minute, Eve becomes the candidate actually elected. The adversary has in effect pulled off a "bait-and-switch" scam on the electorate, in which Alice is the bait and Eve is the switch.

Hazards potentially affecting all voters: Even barring such admittedly-rare "dark horse" election scenarios, as mentioned earlier, insufficiently careful *use* of the E-voting system's write-in capability could compromise the system's security – not just for voters who (intend to) cast write-in ballots, but potentially for *all* voters.

For example, the current documents do not specify how ballots that *appear to* contain write-in options are to be validated and counted after decryption. For example, if a decrypted ballot appears to contain *both* a predetermined candidate choice in the main ciphertext *and* a non-empty ciphertext in the corresponding write-in position, is such a ballot considered valid, and if so which choice "wins" in the counting?

If the write-in candidate overrides the predefined choice in such a ballot, then malicious voting clients could silently override the choices of *all* voters with write-in choices, simply by attaching a write-in ciphertext of the adversary's choosing to a ballot containing the predefined choise the voter actually indicated.

Treating ballots containing both a predefined and a write-in choice as invalid would similarly compromise security for all voters, however. Even if malware-controlled voting clients could not undetectably change predefined voting choices to a valid write-in ballot, the malware-controlled voting client could still undetectably convert valid ballots containing only predefined choices into invalid ballots that would be discarded at counting time, in effect silently disenfranchising voters.

It is of course relatively easy to protect the cast-as-intended verifiability of ballots that make only predefined choices: simply require that the predefined choices *always* have precedence, and that the contents of write-in ciphertexts are considered *only* when a corresponding predefined selection index indicates a particular option designating a write-in choice. But even if this is the way the system already works, it needs to be documented as such.

4 Software (Scope 2)

The author performed some inspection of the current software comprising the Swiss Post E-voting system, particularly to gain a better understanding of various details of its operation and questions that arose in other aspects of this analysis. As stated earlier, the author did *not* attempt a systematic, "line-by-line" inspection of the software, but instead remained focused on broader architectural analysis of the overall system. As the author's limited review of the software did not revealed any substantive findings, this section is left empty.

5 Infrastructure (Scope 3)

This section focuses on the SwissPost infrastructure supporting the deployed E-voting system. The observations in this section are based not only on the key documents listed earlier but also on numerous interviews as part of Scope 3 of the audit.

5.1 Voter authentication second factor

The system specification and cryptographic protocol discuss the issue of voter authentication to the voting server only in general, abstract terms. Indeed, as the Protocol specification points out (section 13.1.2 "Authentication in Voting Phase"), authenticating the voting device to the voting server is technically "pointless" according to the abstract threat model because both voting device and voting server are considered untrustworthy and "the adversary could circumvent it at will." In *practice*, however, authentication of voters to the voting server is extremely important, to avoid the massive denial-of-service or voter-

confusion attacks that would be trivial if miscreants could easily impersonate voters at will.

The system architecture document specifies that "Authentication requires the voter's Start Voting Key (SVK) plus an additional authentication factor" (section 6.2.2 "Voting Phase"). This "additional authentication factor" is left unspecified, which may be reasonable since the specific choice may be very implementation-specific and there may be reason to change it or ultimately to support various second authentication factors.

However, it turns out that the current, one and only "standard" choice for this second authentication factor is the voter's birth year or birth date. This choice is appealing of course because it is something most voters know about themselves and are unlikely to forget. However, it is also a problematic choice of second authentication factor because it is something that is likely to be known by all of the voter's (extended) family members, friends, and anyone who has ever attended (or even seen an announcement for) a birthday party for the voter in question. A person's birth date is also a piece of information that stays fixed throughout a person's lifetime and can never be changed like a password (other than by taking the rather large step of creating a false identity with a different birthdate). Thus, it would likely be preferable to choose something else as the second factor for voter authentication.

5.2 Physical access to control components

While the broader issue of control component independence was addressed earlier in Section 2.3 and will not be repeated here, one particular independence concern at infrastructure level that came to light during Scope 3 discussions is that – at least as of that time – the keys controlling physical access to the locked data center cages for *all* four control components were kept (in a lock box) in the office of a single high-ranking SwissPost official. This represents an independence risk, since that official – or anyone able to gain sufficient access that office – might in principle gain physical access to and hence the opportunity to compromise all four control components.

It is useful and desirable to ensure that *any* physical access to *any* control component's locked cage is access-controlled at the highest feasible organizational level. However, a preferable way to achieve this high-level access control, without compromising the physical independence of the control components, would be to arrange for each control component's cage to be physically locked by *two* independent locks, *both* of which must be unlocked in order to open the cage. The keys to one of these locks would be held by the high-level official controlling access to all cages, while the keys to the other lock would be held (*only*) by the members of the team responsible for that particular control component. Thus, access to any control component's cage should and would require the presence and cooperation of both the high-level official *and* the relevant control component's team members.

5.3 Reproducible builds

A significant step forward that the SwissPost E-voting system has already taken is to implement *reproducible builds* [1, 4] of the E-voting system components. Given a particular snapshot of the source code that a system depends on, a reproducible build system always deterministically produces the same binary. This way, anyone else can independently reproduce the build and verify that it indeed represents the (one and only) correct build output given the input source code, and was not (for example) tampered with to insert a back door at binary level. This state-of-the-art practice represents an important, if partial, solution to certain software supply-chain attacks and the famous problem of "trusting trust" [6].

As of the time of the discussions comprising this audit, however, two issues arose in which the current reproducible build practice appear less complete or systematic than would be ideal. The first issue is that the current operational processes do not require that anyone outside of SwissPost actually reproduce and check the reproducible builds using independent build systems, as would be ideal. The second issue is that the current operational processes do not appear to include clear mechanisms to verify, at software installation or election time, that reproducible builds that are *supposed* to be running on the critical production systems are in fact the builds *actually* running on those production systems. These issues are readily fixable, with some cost/benefit tradeoffs of course.

5.3.1 Are the builds independently reproduced?

In the currently-defined operational processes, the only parties that *actually* reproduce the build on a regular ba-

sis – and in particular who reproduce and verify the production binaries – are (different) teams within SwissPost. This internal independence of build reproduction is important and valuable. It would be even better, however, if some party external to SwissPost – such as an independent auditor – would be tasked with *actually* reproducing and verifying the builds using independent build infrastructure (*i.e.,* not that of SwissPost). In the current build practice, an independent "observer" is present, but the observer merely witnesses the build being reproduced by SwissPost personnel on SwissPost's systems; the observer does not actually reproduce the builds.

In principle anyone else outside SwissPost *should* also be able to reproduce and verify these builds. But there is currently appears to be no operational practice or requirement that *anyone in particular* – independent of SwissPost – actually does so. While not necessarily a major weakness, it would be preferable if the standard operational practice included the actual reproducible building of the production binaries by at least one truly-independent external auditor on separate infrastructure.

5.3.2 Are the reproducible builds those actually run?

Even if the build binaries that are *supposed* to be run on a production system are fully public, reproducible, and even independently reproduced, as of the time of this audit there was no strong operational process for verifying that those particular reproducibly-built binaries are in fact the binaries *actually installed and running* during an election event. That is, perhaps the correct production binaries to be run are fully reproducible and public knowledge, but some other (perhaps compromised) binaries might be secretly installed and running on the production system.

The strongest state-of-the-art solution to this problem would be to use hardware-level *remote attestation*, such as the "trusted boot" or "measured launch" technologies available on Intel and other processors. With this practice, the hardware produces a digitally signed attestation of the exact stack of binaries (firmware, operating system, and application) that are running on the production system at its time of use. Anyone can then verify this attestation to check that the production system is in fact running the (reproducibly built) binaries that it is supposed to be running – provided of course the processor or processor vendor's attestation mechanism are not themselves compromised. Deploying these remote attestation mechanisms is non-trivial and may incur significant additional development and deployment costs, however, since not just the application binaries but operating system and firmware images must be measured and attested as well for full effectiveness of the process. Remote attestation mechanisms are extremely platform-specific, as well, so these costs would have to be born for each of the different platforms that are used (*e.g.,* for diversity and independence).

Until it is practical to invest in such a hardware-level approach on all the most critical devices, a next-best approach would be to use operating system-level measures and operational processes to verify (*e.g.,* before each election) that the expected reproducibly-built binaries are actually installed and running. For example, manual or scripted processes might be invoked at superuser level after each software installation/update event and/or before each election to report the hashes (*e.g.,* SHA256) of all the installed binaries, and the manual operational processes should include explicitly checking those hashes. Further, these measures should be designed to ensure that a one-time compromise of application-level software on a critical device cannot readily defeat these checking measures (*e.g.,*, a compromised binary run once arranging to fake the reported hashes of subsequently-installed reproducible binaries). While there is no perfect solution, at ensuring that the installed application-level binaries themselves never need or execute with supervisor privileges, and that the software-installation and hash-reporting mechanisms are protected at superuser level from any application-level compromise, would be a basic recommended best practice.

6 Conclusion

In its current form as of the present audit, the SwissPost E-voting system embodies a solid architecture for E-voting that in most respects meets or leads the state-of-the-art in international practices for the design, development, and evaluation of E-voting systems. This report has identified and discussed a variety of respects in which the current system falls short of what the author considers ideal for such a system, focusing especially on architecture-level or "end-to-end" issues that potentially involve or interact across all three audit scopes. All of the identified is-

sues appear fixable (or at least possible to improve significantly) in various ways. Some issues appear likely fixable relatively quickly and easily, as in the case of unclear or incomplete documentation. Addressing other issues may be much more complex and costly, and may be feasible only in a future generation of the E-voting system. Nevertheless, it is the author's judgment that as imperfect as the current system might be when judged against a nonexistent ideal, the current system generally appears to achieve its stated goals, under the corresponding assumptions and the specific threat model around which it was designed.

References

[1] Jérémy Bobbio (Lunar). Reproducible Builds for Debian. In *FOSDEM*, February 2014.

[2] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[3] Christian Killer and Burkhard Stiller. The Swiss postal voting process and its system and security analysis. In *E-Vote-ID: International Joint Conference on Electronic Voting*, October 2019.

[4] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium*, pages 1271–1287, 2017.

[5] SPIEGEL Staff. Documents Reveal Top NSA Hacking Unit. *SPIEGEL International*, December 2013.

[6] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, August 1984.

[7] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Scalable anonymous group communication in the anytrust model. In *European Workshop on System Security (EuroSec)*, April 2012.

[8] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through Independence-as-a-service. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.