A Declaration of Software Independence

Wojciech Jamroga[1], Peter Y.A. Ryan[1], Steve Schneider[2],
Carsten Schürmann[3], Philip B. Stark[4]

[1] University of Luxembourg
[2] University of Surrey
[3] IT University of Copenhagen
[4] University of California, Berkeley
Authors listed alphabetically

Abstract. A voting system should not merely report the outcome: it should also provide sufficient evidence to convince reasonable observers that the reported outcome is correct. Many deployed systems, notably paperless DRE machines still in use in US elections, fail certainly the second, and quite possibly the first of these requirements. Rivest and Wack proposed the principle of *software independence* (SI) as a guiding principle and requirement for voting systems. In essence, a voting system is SI if its reliance on software is "tamper-evident", that is, if there is a way to detect that material changes were made to the software without inspecting that software. This important notion has so far been formulated only informally.
Here, we provide more formal mathematical definitions of SI. This exposes some subtleties and gaps in the original definition, among them: what elements of a system must be trusted for an election or system to be SI, how to formalize "detection" of a change to an election outcome, the fact that SI is with respect to a set of detection mechanisms (which must be legal and practical), the need to limit false alarms, and how SI applies when the social choice function is not deterministic.

1 Introduction

Using digital technologies in elections opens up possibilities of enriching democratic processes, but it also brings a raft of new and often poorly understood threats to election accuracy, security, integrity, and trust. This is particularly clear with the so-called *DRE*, Direct-Recording Electronic voting machines, deployed widely in the U.S. after the Help America Vote Act (HAVA) of 2002, which passed in the aftermath of the controversial 2000 presidential election. The original DREs recorded, reported, and tallied cast votes using just software, with no paper record. Thus, an error in or change to that software could alter the outcome without leaving a trace.

It might be argued that the software could be analysed and proven to always deliver a correct result given the input votes. In practice, such analysis and testing is immensely difficult and prohibitively expensive. Moreover, access to the code is often restricted due to commercial or legal constraints. And even if the software could be analysed completely and proven correct, there is still the challenge of guaranteeing that the software actually running on all the machines throughout the voting period is the "correct", verified version.

Consequently, for paperless DRE machines, BMDs, and existing Internet voting systems, voters, election officials et al. are required to place total blind confidence in the correctness of the code running on the devices.

Such concerns prompted calls to add a Voter-Verifiable Paper Audit Trial (VVPAT) to DREs, essentially a printer attached to the DRE that prints the voter's choice, in sight of the voter. In principle, each voter can check whether the paper accurately recorded her preferences, and correct the record if not.[5]

An alternative response—piloted but not yet widely adopted for political elections—is cryptographic end-to-end verifiable voting (E2E-V), which provides voters a means to verify that their vote reaches the tally unaltered and is correctly included in the tally. An accessible introduction to such systems can be found at [BHR$^+$17], and a more extensive description at [HR16]

To capture the essential goal of being able to detect whether faulty software altered the outcome while remaining agnostic with respect to the technology employed to achieve that goal (e.g., a paper record or cryptographic methods), [RW06,Riv08] proposed the principle of *software independence*, which seeks to exclude systems for which the trust in the correctness of the outcome requires trusting the software. The original definition is given as follows:

> A voting system is *software-independent* if an (undetected) change or error in its software cannot cause an undetectable change or error in an election outcome.

[RW06,Riv08] also define a stronger requirement, a system that does not require trusting software, and that is resilient to software-caused errors:

> A voting system is *strongly software-independent* if it is software independent *and moreover, a detected change or error in an election outcome (due to change or error in the software) can be corrected without re-running the election.*

Version 2.0 of the U.S. Voluntary Voting System Guidelines [Ele21], adopted 10 February 2021, incorporates the principle of Software Independence:

> 9.1 - An error or fault in the voting system software or hardware cannot cause an undetectable change in election results.

The principle seems very natural and compelling. It clearly rules out paperless DRE machines and—subject to certain assumptions about voter eligibility and the curation of paper ballots—it clearly admits systems based on hand-marked paper ballots supporting manual recounts, risk-limiting audits [Sta08], and other

[5] There is considerable evidence that voters rarely check machine-generated printout and are unlikely to notice that votes were altered. See, e.g., [Eve07,DKM18,BMM$^+$20,HI21].

forms of audits. However, as soon as we start to consider applying it to other systems, such as end-to-end cryptographically verifiable systems, things are less clear. In particular, many of the terms used in the definition require careful interpretation:

- What exactly do we mean by *the system*? Does it include pollworkers? Auditors? Where do we draw the boundaries?
- What exactly is the *software*? Does it include software involved in determining voter eligibility? Auditing software?
- What exactly does it mean to *detect* an error? Is it enough simply to flag a problem, or must evidence be provided that there really is a problem? What kind of evidence? To whom is the evidence available [ADS20]? What rules out systems that always cry "foul", even when the election outcome is correct?
- What do we mean by *outcome*, in particular, where the social choice function is non-deterministic?

All of this motivates a more formal statement of the principle, which is the aim of this paper. This reveals a number of subtleties, notably that the original definition, read literally, does not exclude systems that reject every declared outcome: there is no penalty for false alarms. We argue that while software independence is a necessary property for a system to be able to deliver a verifiable outcome, it is not sufficient. We also stress the distinction between a *system* being *verifiable* and an *election* being *verified.*

We do not here address vote anonymity, receipt-freeness, coercion resistance, and related concerns. We focus just on the issues of detecting and correcting wrong outcomes while controlling false alarms. In practice, of course, great care needs to be taken in designing a system to provide sufficient transparency and generate sufficient evidence without violating privacy requirements.

We should also remark that, while software independence means that we should not have to place blind faith in the correct behaviour of the code, this does not imply that we should do away with all verification and testing of code. The latter is still important to help ensure the smooth running of any election run using the system, but the assurance of the outcome should not depend on the rigor etc. of such measures.

SI is a desirable property, but the use of an SI system does not by itself give the public adequate reason to trust election outcomes. The fact that it is possible to detect malfunctions of the software does not mean that checks will be performed nor that appropriate action will be taken if problems are detected. And errors or corruption may occur outside the software, e.g. breaches of chain of custody, faulty procedures, incorrect electoral roles, etc.

The notion of software independence is related to notions of end-to-end verifiability (E2E-V); we discuss the relationship in Section 3.2.

2 Formalizing Software Independence

In this section we set the ground for a definition that seeks to capture more formally the spirit of the original natural-language definition. We believe it is faithful to the spirit of the original, but as we shall see, the definition reveals some subtleties, and motivates the game-theoretic definition of the notion of *evidence-based elections* [SW12,AS20], presented below.[6]

2.1 Software Independence... of What?

To merit public confidence, a voting system should generate evidence that can be used to check whether it behaved correctly; typically, that involves a tamper-evident record of voters' expressed preferences, to which the social choice function can be applied to check the reported result. That record might be in the form of a well curated paper audit trail, or, as in many E2E-V systems, data (some of which is encrypted) posted to a public bulletin board (ledger). Furthermore, the system should provide for various checks to be performed on this evidence by the stakeholders: voters, observers, candidates etc. Such checks might be performed before the election starts (e.g. verifying that a transparent ballot box is initially empty), during (e.g. Benaloh challenges), or after (e.g. risk limiting-audits, risk-limiting tallies, verification of zero-knowledge proofs, digital signatures etc.). We refer to such checks generically as “audits”.

We consider *software independence as a property of a voting system $\mathcal{P}$ with respect to a set of audits $\mathcal{A}$*. The voting system $\mathcal{P}$ represents all the components and aspects relevant for how the election is run, starting with the voting protocol, including its implementation (software) and deployment (hardware, physical infrastructure), specification of the environment, assumptions about human users, threat models, etc. The set of audits $\mathcal{A}$ captures the notion of “detectability” by providing an abstract representation of the methods available for detecting something is amiss.

We emphasize that it only makes sense to talk about software independence with a particular view of detection methods. For example, a voting system might be SI if a very powerful (and expensive) kind of instrument or audit can be used, but not if the requisite tools and methods are unaffordable, too time-consuming, or not mandated in law or regulation. On the other hand, another voting system might not be SI with respect to any known audit method, yet may become SI if a new forensic method is invented.[7] We elaborate on both aspects of this characterisation below.

2.2 Voting System and Its Software

Let $\mathcal{P}$ be a specification of how the voting protocol should work. This refers to the overall election system, including hardware, software, procedural, and human

[6] The idea of evidence-based elections is that election officials should not only find the correct winner(s), but should also produce convincing public evidence that they found the correct winner(s)—or else admit that they cannot.

[7] E.g., think of what happened to criminal forensics when DNA tests were introduced.

components. More precisely, $\mathcal{P}$ denotes the system running "correct" software, i.e., software that correctly computes the chosen social choice function over the voted preferences of eligible voters. The software, denoted $\mathcal{S}$, is considered a part of the system. However, in an actual execution of the system, $\mathcal{S}$ may be under the control of the adversary. Thus, $\mathcal{S}$ denotes a part of the system on whose correct behaviour we do not want to rely for evidence that the result is correct. In practise, that might comprise more than software. The spirit of the original definition corresponds to taking $\mathcal{S}$ to be the software that records and interprets votes, applies the social choice function to them, and reports an outcome. It does not include software that may form part of the surrounding system, such as software involved in giving each voter the correct ballot, software used to verify voter eligibility (e.g. voter registration systems and electronic pollbooks), or software involved in auditing the results. Nor does it include the behavior of voters, pollworkers, or election officials.

When we want to make the software $\mathcal{S}$ explicit in the voting system $\mathcal{P}$, then we write $\mathcal{P}[\mathcal{S}]$. Note that it is straightforward to generalise our approach to quantify over other parts of the system, e.g., hardware, people, procedures, etc.

The relevant aspects of system $\mathcal{P}[\mathcal{S}]$ are characterized, on an abstract level, by the following sets and functions:

– $m(\mathcal{P})$: a function that returns all the relevant mutations $\mathcal{P}[\mathcal{S}']$ of the voting system $\mathcal{P}$. We consider $\mathcal{P}[\mathcal{S}']$ as a relevant mutation of $\mathcal{P}[\mathcal{S}]$ if $\mathcal{P}[\mathcal{S}']$ can be obtained from $\mathcal{P}[\mathcal{S}]$ through changing only the software of $\mathcal{P}[\mathcal{S}]$, i.e., $\mathcal{S}$. Hardware and processes and protocols must be the same for $\mathcal{P}[\mathcal{S}]$ and $\mathcal{P}[\mathcal{S}']$. The software that can be changed is restricted to the software involved in collecting voter selections (votes), applying the social choice function to the votes, and reporting the results.
– *In*: the set of possible input sequences. Typically, an input sequence will comprise all the votes expressed[8] by the voters. Depending on the level of granularity in our modelling, it may also include other election-related activity, such as voter registration steps, eligibility verification, coercion attempts, generation of cryptographic keys, where and how each vote was cast, etc. It may also include the full expressed preferences of all the voters. In general, $v \in \mathit{In}$ contains much more information than is needed to determine who won.
– Ω: the set of possible election outcomes. Typically, an outcome is either the tally, or the winner(s) of the election. Depending on the level of granularity, it may also include any other publicly available output of the voting system, such as the content of the web bulletin board. We assume that Ω is finite. For example, in a plurality contest with two contestants, A and B, the possible outcomes in Ω might be "A wins," "B wins," and "A and B are tied." If

[8] By *expressed*, we mean what the voter did: the marks the voters make on the paper or the cell they press on a touchscreen. Of course, a confusing user interface—including poor ballot layout—can cause voters' expressed preferences to differ from their intended preferences. See, e.g. [ADS20].

the social choice function breaks ties, then there would be only two possible outcomes: “A wins” and “B wins.”

– $exec(\mathcal{P}[\mathcal{S}], v)$: a function that returns the set of all the possible executions (runs) of system $\mathcal{P}[\mathcal{S}]$ on the sequence of inputs $v \in In$. Any particular election system with a particular input sequence might have a number of possible executions arising from the different choices that can be made at various points of the voting protocol. For example if a voter is required to provide inputs other than just selections (e.g., to decide whether to challenge an encryption, as allowed in some E2E-V protocols), then different possible executions can arise. In practice, there will usually be just one possible execution given $(\mathcal{P}[\mathcal{S}], v)$, but there may be boundary conditions (e.g. tie-breaking, or randomness in transferring votes) where more than one result is possible. Naturally, $exec(\mathcal{P}[\mathcal{S}'], v)$ is the set of all possible executions of the mutated system $\mathcal{P}[\mathcal{S}']$ on the input sequence v.
– $result(E)$: the outcome of the election for execution E. We lift the function to sets of executions X by fixing $result(X) = \{result(E) \mid E \in X\}$. In the case of the correct system $\mathcal{P}[\mathcal{S}]$, we would expect any outcome in $result(exec(\mathcal{P}[\mathcal{S}], v))$ to be a valid result of the election.

Note that the composition $result(exec(\mathcal{P}[\mathcal{S}], v))$ can be seen as a generalisation of a *social choice function*.

2.3 Available Audits

In the process of running the election, including recording, tallying, and broadcasting the election results, the overall voting system $\mathcal{P}[\mathcal{S}]$ generates evidence that can be used to audit the election. The auditing of an election may overlap with, or be completely separate from the voting procedure. The evidence is provided as input to a decision-making process, represented by a function a, which then provides a judgement. The software in a is assumed to be trustworthy. Such an assumption of trustworthiness needs of course to be justified, and this will usually be by arguing that, if its inputs and intended function are public, anyone who wishes to check the correctness of its outputs could write it again from scratch, or a reputable authority such as the Electronic Frontier Foundation (EFF) could provide a reference implementation.

Evidence produced in the election might include voter registration databases, poll books, physical ballots, encrypted choices, cryptographic receipts, public bulletin boards, zero-knowledge proofs (ZKPs), security videos, the condition of physical seals on ballot boxes, chain-of-custody logs, logs from telephone complaint lines or websites that record “anomalies” voters witnessed during the election, etc. The evidence might not include the “plaintext” voter preferences and generally will not include a voter’s actual interaction with a DRE or BMD.

Some evidence generated during the election will be unreliable or unavailable. For instance, paper ballots do not provide reliable evidence of the outcome if they might have been tampered with, replaced, augmented, or lost; or if voter eligibility checks were not sufficiently accurate. In some E2E-V systems, plaintext votes are not available to check the correctness of the outcome; a system designed to allow voters to check that their intent was recorded correctly (e.g., using a VVPAT or through a Benaloh challenge) does not provide public evidence that voter intent was correctly recorded unless there is both evidence about the number of voters who checked, how effectively they checked, and a mechanism by which it would become known that they found errors, if they find errors. It must be also noted that, by the time a preliminary outcome is available, evidence could be lost, altered, or counterfeited; the election officials might have reacted to some detected problems during the election; and that in turn might generate new possibilities for things to go wrong.

Formally, the capability of the voting authorities (possibly together with independent auditors, public observers, or with voters, e.g., in case of mechanisms for voter verification) to detect malfunctioning of the voting system is characterised by the set $\mathcal{A} = \{a_1, a_2, \dots\}$ of available audit procedures. Let T and F denote the truth values of *true* and *false*, respectively. Each element a_i of $\mathcal{A}$ is a function that takes an execution E of the voting system, and returns an audit judgement $a_i(E) \in \{T, F\}$ such that $a_i(E) = F$ only if there is a change or error in the election outcome. (Below we also consider audits that have a random component, and thus have some probability of returning T or F for any given the voting system execution E.) It is required that a_i must be compatible with the voting system, in the sense that the judgment $a_i(E)$ is based entirely on the evidence available in the execution E of the voting system.

For instance, $\mathcal{A}$ might include verifying poll book signatures, comparing the number of pollbook signatures to the number of votes cast in each precinct, a manual audit of results against a paper trail, checking ZKPs, checking whether digital signatures on cryptographic receipts are authentic, reviewing chain-of-custody records, inspecting equipment log files and security videos, etc.

An exemplar a_i might specify, among its branches, "before opening each box of ballots for central counting, check the seal on the box against a photograph of the seal taken in the polling place. If the seal has been disturbed, interview everyone who has had custody of the box since it was sealed, examine every ballot by hand for signs of tampering or forgery, and compare the number of ballots in the box with the number of pollbook signatures."

3 Possibilistic Formulation of Software Independence

The original definition of SI talks about whether a change to the result is always detectable. This is expressed in terms of possibilities rather than probabilities. Here we see how far we can get with expressing SI possibilistically without involving probabilities. We will also show that it is natural to introduce probabilities into the audit process.

3.1 Basic Formulation

Using the notation introduced in Section 2, the property of Software Independence with respect to a particular election input v and audit method a can be expressed as follows:

$$\begin{aligned} & SI_1(\mathcal{P}[\mathcal{S}], v, a) \iff \\ & \quad \forall \mathcal{P}[\mathcal{S}'] \in m(\mathcal{P}[\mathcal{S}]) \ . \\ & \quad \big(\forall E' \in \mathit{exec}(\mathcal{P}[\mathcal{S}'], v) \ . \ \exists E \in \mathit{exec}(\mathcal{P}[\mathcal{S}], v) \ . \ (\mathit{result}(E) = \mathit{result}(E'))\big) \\ & \quad \vee \big(\exists E' \in \mathit{exec}(\mathcal{P}[\mathcal{S}'], v) \ . \ a(E') = F\big). \end{aligned} \tag{1}$$

The formula states that every execution of any mutation of $\mathcal{P}[\mathcal{S}]$ gives a correct result, or else the malfunction is detectable. More precisely, either every execution of a mutation of $\mathcal{P}[\mathcal{S}]$ gives a result that could have been produced by the correct software, or there is some execution that will fail the audit.

Then, Software Independence holds with respect to a set of possible election inputs $v \in \mathit{In}$ and allowable audit procedures $\mathcal{A}$ if there is some audit procedure $a \in \mathcal{A}$ such that SI holds for all possible inputs:

$$SI_1(\mathcal{P}[\mathcal{S}], \mathcal{A}) \iff \exists a \in \mathcal{A} \ . \ \forall v \in \mathit{In} \ . \ SI_1(\mathcal{P}[\mathcal{S}], v, a). \tag{2}$$

Arguably, formula (1) captures software independence of particular *election*, given the set of votes and actual audit strategy used in the election. In contrast, formula (2) expresses software independence of the *voting system* defined by the voting infrastructure and the available audit strategies.

Remarks. Formulas (1)–(2) capture a rather weak notion of Software Independence. First, they only say that $\mathcal{P}[\mathcal{S}']$ cannot undetectably add incorrect outcomes to the set of possible results of the election. However, a software mutation *removing* some of the correct results may as well satisfy the conditions. We address this issue in Section 3.5.

Secondly, the formalisation is based on a weak notion of detectability. The conditions require that significant software mutations *might* be detected (i.e., they are detected on some possible executions), but there is no guarantee that they can be detected for every execution that produces incorrect outcomes.

A stronger definition of SI is obtained by replacing the right hand side of the disjunction (1) as follows:

$$\forall E' \in \mathit{exec}(\mathcal{P}[\mathcal{S}'], v) \ . \ \big((\exists E \in \mathit{exec}(\mathcal{P}[\mathcal{S}], v) \ . \ \mathit{result}(E) = \mathit{result}(E')) \vee (a(E') = F)\big).$$

This removes the existential quantification over executions and brings E' under the universal quantification. The first formalisation allows for some executions of a mutation not to be caught by an audit even if they give the wrong result. This stronger formalisation states that any execution of a mutation that does not give the correct result should be caught by an audit.

Note also that our formalisation is focused on the potential irregularities due to software mutations. Thus, disturbances of the election outcome due to failures of hardware, dishonest voter behaviour, etc., must only be handled in $\mathcal{P}[\mathcal{S}']$ if they would be caught and dealt with in the ideal system $\mathcal{P}[\mathcal{S}]$.

Audit Strategies. We recall that the characterisation of $\mathcal{A}$ encapsulates the audit methods that are allowable. Considerations as to what should be allowable can include what is possible, affordable (in terms of cost or time), legal (to fit with local election law, preserve the anonymity of votes, etc.), and other considerations as appropriate to the situation. Identifying the limits of what is allowable is itself part of the consideration as to whether a system is software independent. From a technical point of view, the definition of $\mathcal{A}$ will also need to depend on the evidence provided explicitly by the voting system. Thus the formalisation of possible executions E also constrains the audits that are possible, because a is a function on executions: two runs giving rise to the same execution record E must give the same result on audit. For example, if the only evidence collected for audit is the set of paper ballots, then forensic analysis of the hard disks of the voting machines is outside the scope of audit. Conversely if the audit includes the possibility of such analysis, then the evidence provided by an election run should include the relevant state of the hard disks to enable the audit function to be defined.

The sanity condition (or soundness) on an auditing mechanism a for system $\mathcal{P}[\mathcal{S}]$ is that any correct execution of the ideal system will verify positively:

$$\mathbf{sound}(a, \mathcal{P}[\mathcal{S}]) \iff \forall v \in \mathit{In} \;.\; \forall E \in \mathit{exec}(\mathcal{P}[\mathcal{S}], v) \;.\; a(E) = T.$$

Although this is not stated explicitly within the original definition of Software Independence, correct election outcomes should not be flagged by the audit as incorrect, so we will require that every a function in $\mathcal{A}$ be *sound*.

3.2 Relationship to End-to-End Verifiability

The definitions above enable us to highlight an important distinction between Software Independence and End-to-end Verifiability (E2E-V), cf. [BRR$^+$15] for an introduction and [KTV11] for a well-known formalisation. In particular, in a description of a system $\mathcal{P}[\mathcal{S}]$ the component $\mathcal{S}$ explicitly represents only the software, and the context $\mathcal{P}$ remains unchanged. This amounts to requiring that the context $\mathcal{P}$ is trusted in the characterisation of SI. However, when we consider whether the system $\mathcal{P}[\mathcal{S}]$ is end-to-end verifiable, we consider this question with respect to the entire system.

We should note that not all formulations of E2E-V in the literature actually imply correctness of the outcome. Early formulations focused on the ability to detect the corruption of any vote between casting and input to the tally function. To achieve guarantees of correctness we also need measures to prevent ballot stuffing and ballot collisions. Taken together, these imply a bijection between the set of cast votes and the set of votes input to the tally. Here we assume a definition that does encompass these requirements, as does [KTV11]. Here they refer to such a strengthened notion, that does imply correctness if all verification steps give true, as *global verifiability.*

To illustrate the difference, consider the following toy example, which shows that SI does not imply E2E-V: A voting system consists of a ballot box for paper ballots, a scanner, and a software component $\mathcal{S}$ that controls the scanner, interprets the scans, applies the social choice function to the votes, and reports the result. There is a trusted individual $\mathcal{I}$ (appointed by the Election Authority, say) who will also play a key part. A description of the system formulated as $\mathcal{P}[\mathcal{S}]$ would include $\mathcal{I}$ within the definition of $\mathcal{P}$.

Voting: To vote, voters fill out their ballot form, run it through the scanner, then drop it in the ballot box.

Tallying: At the end of the election, $\mathcal{I}$ privately counts the votes from the ballot box and calculates the result r_1. The electronic component $\mathcal{S}$ computes the result r_2 from the scans, and provides this result to $\mathcal{I}$, who privately checks whether $r_1 = r_2$. If so, then $\mathcal{I}$ reports the result. Otherwise an alarm is raised and an audit occurs, consisting of comparing r_1 and r_2. if they are distinct then the audit returns the value F.

The system $\mathcal{P}[\mathcal{S}]$ is SI, because an undetected change in $\mathcal{S}$ cannot undetectably change the result, and the system meets the definition in Line 1. Given a change to the software, either the resulting software still gives the same result, or the audit will return the value F. Note that this relies on the honesty and correct behaviour of $\mathcal{I}$; this is assumed for the characterisation of SI.

The system $\mathcal{P}[\mathcal{S}]$ is not E2E-V. Voters are not able to check that their vote is included in the tally, and there is no check for independent observers that the tally is computed correctly. In particular, $\mathcal{I}$ can simply report a different result and not raise the alarm.

One key difference is that for SI, any part of the system that is not the software is presumed to be acting as it should. Hence, the question is whether a change to $\mathcal{S}$ can change the result when $\mathcal{P}$ behaves correctly.

On the other hand for E2E-V we also consider that $\mathcal{P}$ can behave dishonestly. So $\mathcal{P}[\mathcal{S}]$ is not E2E-V: it is possible for the wrong result to be reported without any verification checks showing incorrect behaviour.

A further distinction is that SI makes no mention of who does the "detecting," whereas E2E-V is quite explicit: each voter can perform the individual check and anyone can perform the universal check. The example above illustrates this point, too.

E2E-V $\Rightarrow$ SI: Conversely, we can reason informally that E2E-V implies SI, via a contrapositive argument as follows. If a system with verification mechanisms is not SI, then by Definition 2 for some input v there is a change to the software $\mathcal{S}'$ that can result in an execution E' with an incorrect result $result(E')$ that passes every audit $audit \in \mathcal{A}$, i.e. it produces an undetectable change to the result. But if the incorrectness of the result is undetectable, then the verification mechanisms cannot detect this, and hence will verify an incorrect result. But this means the system is not E2E-V, since E2E-V requires that if all potential verification steps pass[9] then the result is correct. Note that here we are assuming a strong notion of verifiability, such as global verifiability.

Observe that both audits and verifications can raise an alarm even when the result is correct. We are not concerned with this case in this section, but rather

[9] I.e., every voter checks what individual voters can check (individual verifiability), someone checks the aggregation of votes (universal verifiability), and someone checks that every vote has come from a different eligible voter (eligibility verifiability).

the converse case where the audits and verifications do not raise the alarm even though the result is incorrect.

3.3 SI with Adaptive Audits

The formalization of SI by formulas (1)–(2) assumes that there exists a single audit strategy in $\mathcal{A}$ that can detect malfunction and/or tampering with the voting software. Another option is to swap the quantifiers, and assume that different audit procedures may be applicable on different runs of the voting system (e.g., against different kinds of threats). Now, SI with respect to a set of available audits becomes:

$$\begin{aligned}
&\mathit{SI}_2(\mathcal{P}[\mathcal{S}],\mathcal{A}) \iff \forall v \in \mathit{In} \,.\, \mathit{SI}_2(\mathcal{P}[\mathcal{S}],v,\mathcal{A});\\
&\mathit{SI}_2(\mathcal{P}[\mathcal{S}],v,\mathcal{A}) \iff \\
&\ \forall \mathcal{P}[\mathcal{S}'] \in m(\mathcal{P}[\mathcal{S}]) \,.\\
&\ \big(\forall E' \in \mathit{exec}(\mathcal{P}[\mathcal{S}'],v) \,.\, \exists E \in \mathit{exec}(\mathcal{P}[\mathcal{S}],v) \,.\, (\mathit{result}(E) = \mathit{result}(E'))\big)\\
&\quad \vee \big(\exists E' \in \mathit{exec}(\mathcal{P}[\mathcal{S}'],v) \,.\, \exists a \in \mathcal{A} \,.\, a(E') = F\big).
\end{aligned} \tag{3}$$

That is, either every execution of any mutation of $\mathcal{P}[\mathcal{S}]$ gives a result that could have been produced by the correct software, or there is some execution that will fail at least one audit procedure in the available audit set. Again, formula (3) captures software independence of an election, and (2) expresses SI of the voting system. Note that these notions of detection are still somewhat weak in that they do not ensure that anyone can tell *which* $a \in \mathcal{A}$ suffices for any particular execution E.

3.4 A Refinement

Audit procedures are often nondeterministic by design (e.g., audits that inspect a random sample of ballots, including risk-limiting audits). In our definition of SI, it can be beneficial to separate the randomness of the audit from randomness in the rest of the system. This view can be incorporated by treating audit procedures as functions on system executions E that return a probability distribution on $\{T, F\}$.

For example, for statistical audit of the paper trail, different audit runs result from inspecting different random samples of ballots, each of which has some probability; for some runs, the audit might return T and for others F.

The soundness sanity condition on the auditing mechanism a stays as before.

Having separated the audit non-determinism from the system non-determinism, we can now redefine "undetectable change" to apply to those *system runs* for which the probability that the audit returns F is zero. Let Pr denote probability computed with respect to the audit, treating . Now, software independence of system $\mathcal{P}[\mathcal{S}]$ with respect to the audit set $\mathcal{A}$ becomes:

$$\begin{aligned}
&SI_3(\mathcal{P}[\mathcal{S}], \mathcal{A}) \iff \exists a \in \mathcal{A} \,.\, \forall v \in In \,.\, SI_3(\mathcal{P}[\mathcal{S}], v, a); && (4)\\
&SI_3(\mathcal{P}[\mathcal{S}], v, a) \iff && (5)\\
&\quad \forall \mathcal{P}[\mathcal{S}'] \in m(\mathcal{P}[\mathcal{S}]) \,.\, \forall E' \in exec(\mathcal{P}[\mathcal{S}'], v) \,.\\
&\quad (\exists E \in exec(\mathcal{P}[\mathcal{S}], v) \,.\, result(E) = result(E')) \vee\\
&\quad Pr(a(E') = F) > 0.
\end{aligned}$$

The definition can be equivalently phrased as follows. Let

$$\begin{aligned}
AccResults(\mathcal{P}[\mathcal{S}'], a, v) = \{\omega \mid \exists E' \in exec(\mathcal{P}[\mathcal{S}'], v) \,.\\
(\omega = result(E') \wedge Pr(a(E') = T) = 1)\}
\end{aligned}$$

be the set of *surely accepted results* for $\mathcal{P}[\mathcal{S}']$ on v. That is, these are the possible outcomes of running $\mathcal{P}[\mathcal{S}']$ on input v for which the audit has zero probability of reporting that the outcome is wrong. Note that, for the ideal system $\mathcal{P}[\mathcal{S}]$, if the audit meets the soundness condition this is just the set of possible (correct) outcomes, i.e., $AccResults(\mathcal{P}[\mathcal{S}], a, v) = \{result(E) \mid E \in exec(\mathcal{P}[\mathcal{S}], v)\}$. Since in that case the set does not depend on the audit strategy, we will often write $AccResults(\mathcal{P}[\mathcal{S}], v)$ instead of $AccResults(\mathcal{P}[\mathcal{S}], a, v)$. Then, formula (5) can be rephrased as:

$$\begin{aligned}
&SI_3(\mathcal{P}[\mathcal{S}], v, a) \iff\\
&\quad \forall \mathcal{P}[\mathcal{S}'] \in m(\mathcal{P}[\mathcal{S}]) \,.\, AccResults(\mathcal{P}[\mathcal{S}'], a, v) \subseteq AccResults(\mathcal{P}[\mathcal{S}], v).
\end{aligned}$$

3.5 Software Resilience

The above definition says that every execution of $\mathcal{P}[\mathcal{S}']$ either simulates a legitimate execution of $\mathcal{P}[\mathcal{S}]$ or has a strictly positive chance of being "detected" by the audit. This kind of property is arguably closest to the spirit of the proposal by Rivest and Wack. Also, it corresponds to the intuition that, usually, the only evidence that one has to determine a property of an election system comes from the actual run of the system during the actual election. However, as a system property, it is rather weak. Ideally, one would also like to guarantee the "vice versa" condition, saying that every outcome of the ideal software can be produced by the mutation $\mathcal{P}[\mathcal{S}']$. That is, $\mathcal{P}[\mathcal{S}']$ not only does not introduce any illegal winners, but also does not remove any legally possible ones. Then, every mutation $\mathcal{P}[\mathcal{S}']$ must produce exactly the same set of acceptable election outcomes as the ideal system $\mathcal{P}[\mathcal{S}]$. We call the new property *software resilience (SR)*, and define it formally as follows:

$$\begin{aligned}
&SR(\mathcal{P}[\mathcal{S}], \mathcal{A}) \iff \exists a \in \mathcal{A} \,.\, \forall v \in In \,.\, SR(\mathcal{P}[\mathcal{S}], v, a);\\
&SR(\mathcal{P}[\mathcal{S}], v, a) \iff\\
&\quad \forall \mathcal{P}[\mathcal{S}'] \in m(\mathcal{P}[\mathcal{S}]) \,.\, AccResults(\mathcal{P}[\mathcal{S}'], a, v) = AccResults(\mathcal{P}[\mathcal{S}], v).
\end{aligned}$$

In other words, $SR(\mathcal{P}[\mathcal{S}], v, a)$ requires that every mutation $\mathcal{P}[\mathcal{S}']$ is trace-equivalent to $\mathcal{P}[\mathcal{S}]$ with respect to the surely accepted election outcomes that they can produce.

In practice of course, what the electorate needs is a way to determine, as the end of a given election, whether the reported outcome was not only one of the possible correct outcomes, but also fair in some sense. Where the outcome is uniquely defined this is fine: it is enough that we can determine that it was correct. Where the outcome is not uniquely defined, for example in the event of a tie in a simple plurality vote resolved by the system's software (rather than, for instance, by a public coin toss), this is more delicate: we would like to be able to establish that no possible outcomes were excluded by that particular software running at the time. If the tie is resolved by the software, there is no way to establish one the basis of observation of a single run.

In order to resolve such situations it seems necessary to externalise the mechanism that makes the choice amongst possible outcomes, for example based on a publicly observable coin toss or equivalent. How to provide a truly random source that cannot be predicted or influenced by any way is a topic in its own right, outside the scope of this paper.

Another approach is to regard the outcome as the raw tally, and the resolution of any ties etc. to be outside the scope of the definition. However, the outcome can be correct even when the tally is not—indeed, this is why risk-limiting audits can be efficient. Machine tallies of hand-marked paper ballots are rarely if ever perfectly accurate.

Moreover, non-determinism may be buried in the tabulation algorithm itself, and so not neatly separable. This is for instance the case in the STV variant used in New South Wales, Australia, as well as the D'Hondt method of allocating seats in the parliament in many European countries.

3.6 Thought Experiment

A simple voting system with rather a weak audit highlights some aspects of the definitions.

Consider a voting system $\mathcal{P}_{weak}$ defined as follows:

Voting

1. Votes are cast on paper (filling in a bubble by hand), scanned, and then deposited into a ballot box. The scans are linked to the corresponding paper ballots in a way that allows the scan corresponding to a particular ballot to be retrieved, and vice versa.
2. All of the scans are then published, and the result declared.

Here the software $\mathcal{S}$ controls the scanning, tabulation, and reporting. We assume that there is good physical security of the ballots, and that the total number of ballots is known.

Audit

1. Auditors check whether the number of scans matches the number of ballots. If not, the audit returns F.
2. Auditors inspect every scan and tabulate the resulting interpretation of the votes to obtain an electoral outcome. If that outcome differs from the reported outcome, the audit returns F.
3. A paper ballot is selected at random. Its corresponding scan is retrieved and checked to see whether the human interpretation of that scan matches the human interpretation of the ballot. If not, the audit returns F.

According to the Rivest/Wack definition of SI this system is SI, because any change in the result (caused by a change in the software) can be in principle detected. Thus, it meets the formal characterisation in Line 1. However, this audit may have a low probability of detecting an attack that alters or substitutes scans. If the fraction of the altered scans is δ, then δ is also the chance of detecting the attack. (Moreover, this audit may produce false alarms: the reported outcome could be correct even if some scans were altered.)

3.7 Software Independence for Probabilistic Audits

The thought experiment illustrates that audits can be (and usually are) probabilistic. Although the Rivest/Wack definition of software independence is expressed in possibilistic terms, a comment (almost in passing) in [Riv08] indicates that in practice there should be a high probability of detecting software misbehaviour:

The detection of any software misbehavior does not need to be perfect; it only needs to happen with sufficiently high probability, in an assumed ideal environment with alert voters, pollworkers, etc.

This is a rather stronger requirement, and introduces probability into the characterisation. Where should this probability be introduced?

The idea should be that whatever mutation of $\mathcal{P}$ is considered, and for any execution of that mutation, if the result has been changed then this should be detectable with high probability. The 'detectable' element of this definition is the responsibility of the audit function.

Then we can adjust the definition of Software Independence of Section 3.4 to incorporate the additional requirement that when the result has been changed, the audit has a probability $p_0 > 0$ to notice that:

$$\begin{aligned}
&SI_4(\mathcal{P}[\mathcal{S}], \mathcal{A}, p_0) \iff \exists a \in \mathcal{A} \,.\, \forall v \in In \,.\, SI_4(\mathcal{P}[\mathcal{S}], v, a, p_0);\\
&SI_4(\mathcal{P}[\mathcal{S}], v, a, p_0) \iff\\
&\quad \forall \mathcal{P}[\mathcal{S}'] \in m(\mathcal{P}[\mathcal{S}]) \,.\, \forall E' \in exec(\mathcal{P}[\mathcal{S}'], v) \,.\\
&\qquad (\exists E \in exec(\mathcal{P}[\mathcal{S}], v) \,.\, result(E) = result(E'))\\
&\qquad \lor Pr(a(E') = F) \geq p_0.
\end{aligned}$$

This is clearly stronger than the previous definition in Equations (4)–(5).

4 Probabilistic/Game-Theoretic Definition

In the previous section, we proposed a possibilistic definition of software independence. It was based on the assumption that we can quantify over possibilities (possible mutations of the software, executions of the system, etc.) but cannot formulate constraints with respect to quantitative measures over the possibilities (e.g., probability of executions or computational complexity of a mutation strategy). The first step towards a more quantitative approach was discussed in Section 3.7 where we considered audits with a random component. Here, we present a full-blown quantitative definition of SI. We assume the following:

1. The execution of $\mathcal{P}[\mathcal{S}]$ on an input v defines a probability distribution over all the possible runs in $exec(\mathcal{P}[\mathcal{S}], v)$;
2. The execution of audit method a given a system execution E defines a probability distribution on $\{T, F\}$;
3. The choice of a software mutation belongs to a potentially malicious "attacker," whereas the auditing method is selected by the "defender." The input sequence $v \in In$ is chosen by Nature;
4. The defender must select the audit without knowing the mutation the attacker selected. (However, the audit procedure can be adaptive.) The attacker knows the defender's audit strategy in advance, but not any random elements involved in that strategy. E.g., the attacker might know that the auditor will examine a random sample of ballots, but does not know which particular ballots will be examined.

4.1 Terminology and Notation

As before, Pr denotes probability. Moreover, we will use $\mathsf{Exec}(\mathcal{P}[\mathcal{S}], \mathsf{v})$, $\mathsf{Res}(\mathsf{E})$, and $\mathsf{Aud}(\mathsf{E})$ for the random variables ranging over possible runs $E \in exec(\mathcal{P}[\mathcal{S}], v)$, possible election outcomes $\omega \in result(E)$, and audit judgments in $\{T, F\}$, respectively.

Election Environment. Given the input $v \in In$ (in particular, the voters' expressed preferences), the voting system $\mathcal{P}[\mathcal{S}]$ defines a probability distribution $Pr(\mathsf{Exec}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = E)$ over the possible runs $E \in exec(\mathcal{P}[\mathcal{S}], v)$. Similarly, given a run E of the voting system, $Pr(\mathsf{Res}(\mathsf{E}) = \omega)$ denotes the probability that the election outcome is $\omega \in \Omega$. Note that the social choice function can be now represented by the probability distribution

$$Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = \omega) = \sum_{E \in exec(\mathcal{P}[\mathcal{S}], v)} Pr(\mathsf{Exec}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = E) \cdot Pr(\mathsf{Res}(\mathsf{E}) = \omega).$$

Deterministic social choice functions amount to randomized functions that put all their mass on a single $\omega \in \Omega$.

For instance, in a two-candidate plurality contest with ties broken at random, the set of outcomes can be defined as $\Omega = \{a, b\}$ with a standing for "Alice wins" and b for "Bob wins." If the election input $v \in In$ contains more votes for Alice than for Bob, then $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = a) = 1$ and $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = b) = 0$. If v contains more votes for Bob than for Alice, then $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = a) = 0$ and $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = b) = 1$. If v has the same number of votes for Alice and Bob, then $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = a) = Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = b) = \frac{1}{2}$.

If an election produces outcome ω that has probability zero, that is, if $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = \omega) = 0$, then the outcome is presumptively incorrect.[10] For a single election, if $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = \omega) > 0$, we cannot tell whether $\mathcal{P}[\mathcal{S}]$ assigns the *correct* probability to ω: that would require replicating the execution. Hence, we consider an outcome ω to be *admissible* for $\mathcal{P}[\mathcal{S}]$ and v if the probability of that outcome is strictly positive, that is, if $Pr(\mathsf{Res}(\mathcal{P}[\mathcal{S}], \mathsf{v}) = \omega) > 0$ (the outcome is expected to occur sometimes for that vote profile and that social choice function). We denote the set of such outcomes by $\mathcal{W}_{\mathcal{P}[\mathcal{S}],v}$.

Attack and Defense Strategies. We model the interplay between threats (regardless of their cause) and mitigations as the election unfolds by means of two *strategies* that play against each other: an *attack strategy* and a *defense strategy.*

An *attack strategy* f interferes with the ideal operation of the election by changing the "software" of the election system. (Recall that we use the term "software" abstractly, to denote those things under consideration that might behave incorrectly, which might include more than computer code, depending on context.) Each f amounts to a (possibly randomized) plan that specifies the action that the attacker will take if a given circumstance occurs. It involves the vulnerabilities and failure modes of the overall election, and represents how outcomes and evidence might be altered by failures or adversarial attacks. The involved software mutations are drawn from $m(\mathcal{P}[\mathcal{S}])$. The input v is the set of "true" votes of the eligible voters.

We denote the set of *feasible attack strategies* by $\mathcal{F}_{m(\mathcal{P}[\mathcal{S}])}$. Note that such strategies may have to satisfy some constraints. For instance, it might not be computationally feasible to fake a ZKP. Or it might not be possible to alter marks on paper ballots undetectably, to steal a ballot box and its contents undetectably, or to corrupt a multipartisan group of auditors into faking audit results.

A *defense strategy* g conducts tests and countermeasures to judge whether the announced outcome of the election is correct. Each g amounts to a (possibly randomized) conditional plan that specifies the actions the defender will take in a given set of circumstances. Defense strategies consist of actions that the "checkers" (elections officials, auditors, public, etc.) can take before, during, and after the election to try to ensure that the outcome is correct, and to assess whether the outcome is correct, despite the fact that things might have gone wrong—that is, despite f. Clearly, they can have random elements, such as statistical audits. Given an election run E, $Pr(\mathsf{Aud}(\mathsf{g}, \mathsf{E}) = AJ)$ is the probability that the defense strategy g returns audit judgment $AJ \in \{T, F\}$ on E. The set of possible defense strategies based on audit methods $\mathcal{A}$ is denoted by $\mathcal{G}_{\mathcal{A}}$ The set $\mathcal{G}_{\mathcal{A}}$ is fixed after $\mathcal{F}_{m(\mathcal{P}[\mathcal{S}])}$ is known, but before the apparent outcome ω is known, and without knowledge of f. That is, methods for assessing the outcome may

[10] Recall that the set of outcomes is assumed to be finite.

depend on the kind of evidence the system generates, the ways the ideal evidence might be corrupted, and the execution trace E, including reported tallies and outcomes. The strategies in $\mathcal{G}_{\mathcal{A}}$ must satisfy legal and practical constraints, as discussed above.

Both f and g are "interactive," in the sense that the actions taken under a particular g can depend on circumstances generated by the actions under f, and *vice versa*, as well as on random elements. The defense strategy is restricted to the "audits"; the attacker has no influence on audits other than through $\mathcal{S}$.

Execution Semantics for Strategies. The choice of attack (f) and defense (g) strategies determine how probable different election runs are, which in turn affects the chance that the audit identifies incorrect outcomes. We model this through the probability distribution $Pr(\mathsf{Exec}(\mathcal{P}[\mathcal{S},\mathsf{f}],\mathsf{v}) = E)$ on the set of system executions, for system software $\mathcal{S}$, attack strategy f, and input votes v. For any given g, this induces a probability distribution on the audit decisions $\mathsf{Aud}(\mathsf{g},\mathsf{E})$. Now,

$$Pr(\mathsf{Aud}(\mathsf{f},\mathsf{g},\mathsf{v}) = AJ \mid \mathcal{W}) = \\ \sum_{\omega\in\mathcal{W}} \sum_{E\in exec(\mathcal{P}[\mathcal{S},f],v)} Pr(\mathsf{Exec}(\mathcal{P}[\mathcal{S},\mathsf{f}],\mathsf{v}) = E)\cdot Pr(\mathsf{Res}(\mathsf{E}) = \omega)\cdot Pr(\mathsf{Aud}(\mathsf{g},\mathsf{E}) = AJ)$$

denotes the probability that the announced outcome will be accepted (for $AJ = T$) or rejected (for $AJ = F$), given that the announced outcome is in $\mathcal{W}$.

As in Section 3, we take v to be fixed when defining software independence of a particular *election*. Moreover, we are interested in $\mathcal{W} = \{\omega\}$, where ω is the outcome that has been announced. In defining software independence of an *election system*, we quantify over the possible election inputs $v \in In$, and do not condition on $\mathcal{W} = \{\omega\}$.

4.2 Game-Theoretic Definition of SI

We will cast software independence in terms of a game, in a manner analogous to how semantic security of cryptographic algorithms is captured, or to how estimation problems are formalized in statistical decision theory. An election is seen as a strictly competitive game between the adversary choosing an attack strategy $f \in \mathcal{F}_{m(\mathcal{P}[\mathcal{S}])}$ and the checker choosing a defense strategy $g \in \mathcal{G}_{\mathcal{A}}$. The payoffs of the checker are multicriterial (and thus only partially ordered), and given by the respective probabilities of false positive and false negative output of the audit procedure. The solution concept is based on *minimax*, i.e., the checker minimizes the loss assuming the worst case (most damaging) of the adversary. (Since the payoff is multicriterial, there is no minimax strategy *sensu stricto*, but the analysis is worst-case.) Moreover, the adversary is assumed to adapt the attack strategy f to the defense strategy g selected by the checker. On the other hand, the checker must choose the defense strategy without knowing the attack strategy.

Formally, given a defense strategy $g \in \mathcal{G}_{\mathcal{A}}$, an election input $v \in In$, and a set of admissible election outcomes $\mathcal{W}_{\mathcal{P}[\mathcal{S}],v}$, we define two kinds of costs that the checker wants to minimize:

$$\begin{aligned}
\epsilon(g,v) &= \sup_{f \in \mathcal{F}_{m(\mathcal{P}[\mathcal{S}])}} Pr(\mathsf{Aud}(\mathsf{f},\mathsf{g},\mathsf{v}) = F \mid \mathcal{W}_{\mathcal{P}[\mathcal{S}],v}),\\
\delta(g,v) &= \sup_{f \in \mathcal{F}_{m(\mathcal{P}[\mathcal{S}])}} Pr(\mathsf{Aud}(\mathsf{f},\mathsf{g},\mathsf{v}) = T \mid \overline{\mathcal{W}}_{\mathcal{P}[\mathcal{S}],v})\\
&= 1 - \inf_{f \in \mathcal{F}_{m(\mathcal{P}[\mathcal{S}])}} Pr(\mathsf{Aud}(\mathsf{f},\mathsf{g},\mathsf{v}) = F \mid \overline{\mathcal{W}}_{\mathcal{P}[\mathcal{S}],v}).
\end{aligned}$$

That is, ϵ is the largest chance that the checker rejects an admissible outcome (*false negative*), and δ is the largest chance that he fails to reject an inadmissible outcome (*false positive*).

Definition 1 ((ϵ, δ)-SI). *Consider an election where v was the actual input and g the used defense strategy. The election is* (ϵ_0, δ_0)-software independent *if $\epsilon(g, v) \leq \epsilon_0$ and $\delta(g, v) \leq \delta_0$, i.e., the probability of false negative is bounded by ϵ_0, and the probability of false positive is bounded by δ_0.*

Moreover, the voting system *is (ϵ_0, δ_0)-software independent if there exists $g \in \mathcal{G}_{\mathcal{A}}$ such that for all $v \in In$, the resulting election is (ϵ_0, δ_0)-SI.*

Ideally, elections should be fully reliable. This motivates the following definition.

Definition 2 (Strict SI). *An election (respectively, voting system) is* strictly software independent *if it is $(0, 0)$-software independent.*

Unfortunately, strict SI might be hard to achieve in realistic scenarios. In that case, we should at least require that the defense strategy is more effective than random guessing. Suppose that the checker tosses a biased coin (independently of all other election processes) that has probability p of landing heads, and then rejects the announced outcome if the coin lands heads and accepts the outcome if the coin lands tails. That rule g_p attains $\epsilon(g_p, v) = p$ and $\delta(g, v) = 1 - p$, so $\epsilon(g_p, v) + \delta(g_p, v) = 1$. By using the available evidence one should be able to do better. This leads to the following definition:

Definition 3 (loose SI). *An election (respectively, voting system) is* loosely software independent *if it is (ϵ, δ)-software independent with $\epsilon + \delta < 1$.*

For example, consider a voting system based on hand-marked paper ballots kept secure and trustworthy, with trustworthy eligibility determinations, subject to a risk-limiting audit with risk limit $\alpha < 1$. Such a voting system is $(0, \alpha)$-SI and loosely SI. If there were an automatic recount instead of a risk-limiting audit, the system would be strictly SI.

5 Conclusions

We have presented several formalisations of the notion of software independence. In doing so we have shown that, like many security properties, this seemingly simple and intuitive notion actually harbours many subtleties. For example we observe that it is important to exclude trivial systems that simply reject all runs of an election. The original definition clearly intended this but did not explicitly require it. Many of the terms used in the definition require precise definition. For example, "detection" should not just mean claiming to have observed a departure from correct behaviour but also to be able to provide evidence that such a departure did indeed occur. This is related the notion of dispute resolution: the ability of a third party to be able to determine whether alarm is genuine or false.

We have enriched our definitions to allow for non-determinism or randomisation in the execution of the protocols, and in particular in the social choice function. Further, we have argued that purely possibilistic definition is not necessarily that useful, rather one should extend that definition to account for the probabilities of detecting erroneous behaviour.

Another insight from our formalisation is the need to precisely define when is meant by the "system" and the "software". By the latter we mean those parts of the system on whose behaviour we do not want the correctness of the outcome to depend. However, for many systems this will not include all the software of the system, for example, the auditing components and procedures may require software and we typically assume that such software is correct with respect to its specification. Such assumptions can typically be justified by arguing that auditing algorithms can typically be rerun on independent implementations, so corruption of an instance of this software is itself detectable.

In future work we plan to apply our definitions to a representative sample of verifiable voting systems. We also plan to generalise the notion of software independence to include other components of the system: hardware, people, procedures etc. This brings us back to the question of defining the boundaries of the sub-system that we require the correctness of the outcome to be independent.

Acknowledgements

Peter Y.A. Ryan would like to thank the FNR (Fond Nationale de Research Luxembourg) and the Velux Foundation for support during his sabbatical and to ITU Copenhagen for hosting him when this work was initiated. Steve Schneider is grateful to EPSRC for funding through the VOLT project EP/P031811/1. Wojciech Jamroga acknowledges the support of the National Centre for Research and Development, Poland (NCBR), and the FNR Luxembourg under the PolLux/FNR-CORE projects VoteVerif (POLLUX-IV/1/2016) and STV (POLLUX-VII/1/2019).

References

[ADS20] A.W. Appel, R. DeMillo, and P.B. Stark. Ballot-marking devices cannot assure the will of the voters. *Election Law Journal, Rules, Politics, and Policy*, 19(3), 2020.

[AS20] A.W. Appel and P.B. Stark. Evidence-based elections: Create a meaningful paper trail, then audit. *Georgetown Law Technology Review*, 4.2:523–541, 2020. `https://georgetownlawtechreview.org/wp-content/uploads/2020/07/4.2-p523-541-Appel-Stark.pdf`.

[BHR$^+$17] M. Bernhard, J.A. Halderman, R.L. Rivest, P. Vora, P.Y.A. Ryan, V. Teague, J. Benaloh, P.B. Stark, and D. Wallach. Public evidence from secret ballots. In R. Krimmer, M. Volkamer, N. Braun Binder, N Kersting, O. Pereira, and C. Schürmann, editors, *Electronic Voting. E-Vote-ID 2017. Lecture Notes in Computer Science, 10615.* Springer, 2017.

[BMM$^+$20] Matthew Bernhard, Allison McDonald, Henry Meng, Jensen Hwa, Nakul Bajaj, Kevin Chang, and J. Alex Halderman. Can voters detect malicious manipulation of ballot marking devices? In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 679–694, 2020.

[BRR$^+$15] Josh Benaloh, Ronald Rivest, Peter YA Ryan, Philip Stark, Vanessa Teague, and Poorvi Vora. End-to-end verifiability, 2015. arXiv:1504.03778.

[DKM18] R. DeMillo, R. Kadel, and M. Marks. What voters are asked to verify affects ballot verification: A quantitative analysis of voters' memories of their ballots. Technical report, 2018.

[Ele21] Election Assistance Commission. Voluntary voting system guidelines VVSG 2.0, 2021. https://www.eac.gov/sites/default/files/TestingCertification/Voluntary_Voting_System_Guidelines_Version_2_0.pdf.

[Eve07] S.P. Everett. *The Usability of Electronic Voting Machines and How Votes Can Be Changed Without Detection.* PhD thesis, Rice University, 2007.

[HI21] A.A. Haynes and M.V. Hood III. Georgia voter verification study. Technical report, 2021.

[HR16] Feng Hao and Peter Y. A. Ryan. *Real-World Electronic Voting: Design, Analysis and Deployment.* Auerbach Publications, USA, 1st edition, 2016.

[KTV11] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, privacy, and coercion-resistance: New insights from a case study. In *32nd IEEE Symposium on Security and Privacy*, pages 538–553, 2011.

[Riv08] R.L. Rivest. On the notion of "software independence" in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.

[RW06] R.L. Rivest and J.P. Wack. On the notion of "software independence" in voting systems (draft version of July 28, 2006). Technical report, Information Technology Laboratory, National Institute of Standards and Technology, 2006.

[Sta08] P.B. Stark. Conservative statistical post-election audits. *Annals of Applied Statistics*, 2008.

[SW12] P.B. Stark and D.A. Wagner. Evidence-based elections. *IEEE Security and Privacy*, 10:33–41, 2012.