Chapter 1

Software Independence Revisited

Ronald L. Rivest
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology (MIT)
Cambridge, MA 02139
`rivest@mit.edu`

Madars Virza
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology (MIT)
Cambridge, MA 02139
`madars@mit.edu`

CONTENTS

1.1 Introduction

Democracy depends on elections, yet elections are complex but fragile processes involving voters, election officials, candidates, procedures and technology. Voting systems are evaluated in terms of their security, usability, efficiency, cost, accessibility and reliability. A good voting system design should be based on sound principles.

The principle of "software independence" was introduced by Rivest and Wack [488] and Rivest [486]:

> 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.

For example, optical scan and some cryptographically-based voting systems are software independent.

Software independence is one form of auditability, enabling detection and possible correction of election outcome errors caused by malicious software or software bugs.

This chapter begins with a review of the definition of software independence as given by Rivest and Wack [488] and Rivest [486]; starting with a review of the issue of software complexity (Section 1.2) and a re-presentation of the definition of software independence and its rationale (Section 1.3). The reader is encouraged to consult the original papers [488, 486] for further details, elaboration and clarification of the original definition.

Further sections discuss evidence-based elections (Section 1.6), end-to-end verifiable voting (Section 1.8) and verifiable computation (Section 1.10).

1.2 Problem: Software Complexity of Voting Systems

We start by describing the problem that software independence addresses: the difficulty of assuring oneself that voted ballots will be recorded accurately by complex and difficult-to-test software in all-electronic voting systems. We emphasize that the problem is providing such assurance: the software may well be correct, but convincing oneself (or others) that this is the case is effectively impossible.

Electronic voting systems are complex and continue to grow more so. The requirements for privacy for the voter, for security against attack or failure and for the accuracy of the final tally are in serious conflict with each other. It is common wisdom that complex and conflicting system requirements lead to burgeoning system complexity.

Voting system vendors express and capture this complexity via software in their voting systems.

As an example, consider a direct-recording electronic (DRE) voting system, which typically provides a touchscreen user interface for voters to make selections and cast ballots, and which stores the cast vote records in memory and on a removable memory card. A DRE may display an essentially infinite variety of different ballot layouts, and may include complex accessibility features for the sight-impaired (e.g., so that a voter could use headphones and be guided to make selections using an audio ballot).

An issue, then, is how to provide assurance, despite the complexity of the software, that the voting system will accurately record the voter's intentions. A pure DRE voting system produces only electronic cast ballot records, which are not directly observable or verifiable by the voter.

Consequently, no meaningful audit of the DRE's electronic records to determine their accuracy is possible; accuracy can only be estimated by a variety of other (imperfect) measures, such as comparing the accumulated tallies to pre-election canvassing results, performing software code reviews and testing the system accuracy before (or even during) the election.

1.2.1 The Difficulty of Evaluating Complex Software for Errors

It is a common maxim that complexity is the enemy of security and accuracy, thus it is very difficult to evaluate a complex system. A very small error, such as a transposed pair of characters or an omitted command to initialize a variable, in a large complex system may cause unexpected results at unpredictable times. Or, it may provide a vulnerability that can be exploited by an adversary for large benefits.

Finding all errors in a large system is generally held to be impossible in general or else highly demanding and extremely expensive. Our ability to develop complex software vastly exceeds our ability to prove its correctness or test it satisfactorily within reasonable fiscal constraints (extensive testing of a voting system's software would certainly be cost-prohibitive given how voting in general is funded). A voting system for which the integrity of the election results intrinsically depends on the correctness of its software will always be somewhat suspect.

As we shall see, the software-independent approach follows the maxim, "Verify the election results, not the voting system."

1.2.2 The Need for Software-Independent Approaches

With the DRE approach, one is forced to trust (or assume) that the software is correct. If questions arise later about the accuracy of the election results (or if a recount is demanded), there is again no recourse but to trust (or assume) that the voting system did indeed record the votes accurately. We feel that one should strongly prefer voting systems where the integrity of the election outcome is not dependent on trusting the correctness of complex software.

The notion of "software independence" captures exactly this desirable characteristic of providing election results that are verifiable, without having to depend on the assumption that the software is correct.

For users of software-independent voting systems, verification of the correctness of the election results is possible. There need be no lingering unanswerable concern that the election outcome was affected or actually determined by some software bug (or worse, e.g., by a malicious piece of code).

1.3 Definition and Rationale for Software Independence

We now repeat the definition of software independence, and explore its meaning.

> 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.

A voting system that is not software independent is said to be *software dependent*—it is, in some sense, vulnerable to undetected programming errors, malicious code, or software manipulation, thus the correctness of the election results is dependent on the correctness of the software.

The first use of "undetected" in the definition is to give emphasis to software faults that are *undetected* not being able to cause *undetectable* changes; it is in parentheses because already-known faults may be dealt with by other means.

The intent of the definition of software independence is to capture the notion that a voting system is unacceptable if a software error can cause a change in the election outcome, with *no evidence available that anything has gone wrong*. A "silent theft" of the election should not be possible with a software-independent system. (At least, not a theft due to software...)

To illustrate the rationale for software independence, let us run a "thought experiment." Put yourself in the place of an adversary and imagine that you have the power to secretly replace any of the existing software used by the voting systems by software of your own construction. (You may assume that you have the original source code for the existing software.)

With such an ability, can you (as the adversary) change an election outcome or "rig an election" without fear of detection?

If so, the system is *software dependent*—the software is an "Achilles heel" of the voting system. Corrupting the software gives an adversary the power to secretly and silently steal an election.

If not, the system is *software independent*—the voting system as a whole (including the non-software components) has sufficient redundancy and potential for cross-checking that misbehavior by the software can be detected. The detection might be by the voter, by an election official or technician, by a post-election auditor, by an observer or by some member of the public. (Indeed, anyone but the adversary.)

In such a "thought experiment," we are considering the adversary as some evil agent that could load fraudulent software into voting systems. More realistically, we may consider this adversary to be an abstraction of the limitations of the software development process and testing process. (As such, for the purposes of determining whether a system is software-independent, one should presume that the software errors were present when the software was written and were not caught by software development control processes or by the certification process.)

As we have stated, complex software is difficult to write and to test, and will therefore contain numerous unintentional "bugs" that occasionally can cause voting systems to report incorrect election results. It would be extremely difficult and expensive to determine with certainty that a piece of software is free of bugs that might change an election outcome. Given the relatively small amounts of funding allocated for developing and testing voting system software, we may safely consider it as *effectively impossible*. Thus, the software itself is not considered evidence of a change in the election outcome for the purposes of the definition of software independence. Such "evidence" is too hard to evaluate.

1.3.1 Refinements and Elaborations of Software Independence

There are a number of possible refinements and elaborations of the notion of software independence. We now motivate and introduce the distinction between *strong software independence* and *weak software independence*.

Security mechanisms are typically one of two forms: *prevention* or *detection*. Detection mechanisms may also be coupled with means for *recovery*. When identification of participants and accountability for actions is also present, then detection mechanisms are also the foundation for *deterrence*. Given the importance of recovery mechanisms in addition to detection mechanisms, we propose the following two refinements of the notion of software independence:

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

A voting system that is *weakly software independent* conforms to the basic definition of software independence but is not strongly software independent—that is, there is no recovery mechanism.

1.3.2 Examples of Software-Independent Approaches

Currently, there are two general categories of software-independent approaches.

Voter-verifiable paper record (VVPR) approaches constitute the first category, since the VVPR allows (via a recount) the possibility of detecting (and even correcting) errors due to software. Accordingly, these voting systems can be strongly software independent.

The most prominent example in this category is the optical scan voting system used by most U.S. voters since the 2006 elections. The paper ballot is voter verifiable because the voter completes the ballot and can attest to its accuracy before it is fed into the optical scanner; the paper ballot thus serves as an audit trail that can be used in post-election audits of the optical scanner's electronic results. An electronic ballot marking system (EBM) may also be used to record the voter's choices electronically with a touchscreen interface and then to print a high-quality voter verifiable paper ballot for feeding into the optical scanner.

Another example in this category is the voter-verified paper audit trail (VVPAT) voting system, similar to a DRE but with a printer and additional logic. It produces two records of the voter's choices, one on the touchscreen display and one on paper (a VVPR). The voter must verify that both records are correct before causing them to be saved.

Cryptographic voting systems constitute the second category of software-independent voting system approaches. They can provide detection mechanisms for errors caused by software changes or errors [43, 143, 150, 340, 418, 501, 503]). At one level, they can enable voters to detect when their votes have been improperly represented to them at the polling site, and a simple recovery mechanism (re-voting) is available. At another level, they can enable anyone to detect when their votes have been lost or changed, or when the official tally has been computed incorrectly. Recovery is again possible. Most of the recently proposed cryptographic voting systems are strongly software independent.

Receipt-based cryptographic voting systems involve a physical, e.g., paper, receipt that the voter can use to verify, during the process of voting, whether his or her ballot was captured correctly. The contents of the receipt, in general, employ cryptography in some form so that the voter is able to verify that the votes were recorded accurately; the receipt does not show how the voter voted.

Approaches to software independence other than pure use of VVPR or cryptographic voting systems are potentially possible, although beyond the scope of our chapter.

1.4 How Does One Test for Software Independence?

This brings up a more subtle point in the definition. What aspects of the voting system make it "software independent?" Is it just the hardware and software, or does it also include the surrounding procedures? For example, is a voting system still software independent if no post-election audits are performed?

The answer is that a voting system is software independent if, after consideration of its software and hardware, it enables use of any election procedures needed to determine whether the election outcome is accurate without having to trust that the voting system software is correct. The election procedures could include those carried out by voters in the course of casting ballots, or in the case of optical scan and VVPAT, they could include election official procedures such as post-election audits.

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.

As an example, consider the EBM which prints out a filled-in optical scan ballot. Some voters may not review the printed ballot at all. Yet the EBM is still software independent; there is a significant probability that software misbehavior by the EBM will be detected (this is similarly true of VVPAT). For the purposes of the definition of "software independence," we assume that (enough) voters are sufficiently observant to detect such misbehavior. (If this assumption were discovered to be false in practice, some increase in voter education might be necessary.) Although some forms of such detectable misbehavior may leave no tangible proof of misbehavior, the definition of software independence does not require that all misbehavior have tangible proof; it is sufficient that the relevant misbehavior be detectable and reportable.

Continuing with this example, we note that there is also software in the optical scanner used to scan the ballots that might produce incorrect output. But such misbehavior is detectable by a post-election audit procedure that hand-counts the paper ballots, thus the optical scan voting system is software independent. (Note that such audits are typically statistical in nature and are thus not perfect detectors of misbehavior. But a well-designed audit will catch such misbehavior with reasonable probability. See [66, 285].

To illustrate further, then, say that no post-election audit of an optical scan-based election is required if the apparent margin of victory is more than 10%. An optical scan system would still be considered software independent in such an election, since the original voter-verified paper ballots are available for review, and software misbehavior can still in principle be detected. (As a side note: we feel that such post-election audits are always a good idea and that “no audit” should not be an option. If an apparent margin of victory is large, a smaller audit is appropriate.)

As a final example, say that electronic pollbook systems are used in an optical scan-based election, but the electronic pollbooks do not create a contemporaneous paper record for each voter. Thus, their software must be trusted to show that the number of optical scan records (paper and electronic) accurately reflect the actual number of voters who used the scanners. Are these systems software independent? We would argue that the answer is no for the electronic pollbook, as the design of this system has prevented an audit to determine if the number of optical scan records is correct, i.e., its software must be trusted to be correct. A contemporaneous paper record would have made the electronic pollbook software independent.

1.5 Discussion

1.5.1 Implications for Testing and Certification

Given the exceptional difficulty of proving software to be correct, it is a reasonable proposal to disallow voting systems that are software dependent altogether.

If testing and certification of software-dependent voting systems are to be nonetheless contemplated, then one should expect the certification process should be very much more demanding and rigorous for a software-dependent voting system than for a software-independent voting system. The manufacturer should submit a formal proof of correctness, with perhaps an assurance level corresponding to EAL level 6 or 7 [1] and public disclosure of the source code. Moreover, the voting system must permit proof it is running the software it is supposed to.

1.5.2 Related Issues

There may be other aspects of software misbehavior that don’t quite fit our proposed notion of software independence. For example, software may bias a voter’s choices in subtle ways (say by displaying one candidate’s name in slightly brighter characters on a touchscreen). These issues fall outside the scope of software independence, since the correct “election outcome” isn’t well-defined until the voter indicates her choice. Software independence is focused on the correctness of the election results, and not on other aspects of the voting process.

Some voting systems, such as certain STV (single transferable vote) systems, determine an election outcome in a way that may be randomized (e.g., for breaking ties). A voting system whose software breaks ties in different ways would not be

[1] `https://en.wikipedia.org/wiki/Common_Criteria`,
`https://en.wikipedia.org/wiki/Evaluation_Assurance_Level`

considered to violate software independence, as long as any outcome so determined is a legally acceptable election outcome given the cast vote records.

It is worth emphasizing that the records produced of voters' choices should be of sufficient quality and durability to be usable in a post-election audit.

1.6 Evidence-Based Elections

Recently (2012), Stark and Wagner proposed [541] the notion of "evidence-based elections," a broad framework for understanding how confidence in election outcomes can be achieved, through a combination of auditability (achieved via strongly software-independent voting systems) and auditing (specifically, compiliance audits and risk-limiting audits).

In this framework, strongly software-independent voting systems generate the audit trail (typically, but not necessarily, consisting of voter-verified paper ballots), while the compliance checks that the audit trail has not been corrupted or compromised, and the risk-limiting audit ensures (by appropriate statistical sampling and analysis) that the audit trail is consistent with the stated election outcome.

It is the combination of auditability and actual auditing that provides the evidence for the correctness of the election outcome. As they put it,

$$\text{evidence} = \text{auditability} + \text{auditing}$$

In this framework, the voting system software is not part of the evidence being evaluated during the audit. Furthermore, as Stark and Wagner argue effectively, trust in the software is not necessary for developing confidence in election outcomes. Indeed, the need to have time-consuming and expensive voting system certifications may be hampering the development of voting systems that enable elections that provide the evidence necessary to have trustworthy outcomes.

We endorse the "evidence-based elections" framework described by Stark and Wagner. Software independence is a necessary component of such a framework.

1.7 The Use of a Public Ledger

The development of the internet has made possible the "democratization" of many capabilities previously reserved for the few. The diversity and quantity of information available for public review, compared to the situation only two decades ago, is quite astonishing.

Of interest here is the availability of *transactional* data generated by users with some information system. Usually such information is made available in the form of a per-application database, maintained by the transaction service provider. Sometimes the transaction data is available only to the user involved in the transaction (e.g., credit-card data); sometimes it is public (e.g., real-estate transactions).

So, one may reasonably ask whether election data can or should be made available online and even made available to the public in the form of a "public ledger" or "public bulletin board."

There is no reason not to do so, except when doing so might violate voter privacy. Making election information available online should not enable voters to sell their votes or be coerced into voting in a certain way.

Indeed, making the audit trail publicly available may engender greater trust in the election outcome, since the public may help with the verification that the audit trail is consistent with their knowledge as to how they voted and with the stated election outcome.

The Bitcoin [403] block chain exemplifies an extreme position with respect to democratization: not only is the transaction ledger totally public, but the ledger is maintained without trusted third parties by a clever peer-to-peer mechanism based on incentives for "miners" who extend the block-chain containing the ledger by solving cryptographic puzzles.

Yet, while proposals have been floated, and even tested, for block-chain based voting,[2] it is important to distinguish the questions "(1) What information is on the audit trails?" and "(2) Is that information public?" The use of a public ledger (as, for example, provided by block-chain based ledger) provides an affirmative answer to (2), it does nothing to answer (1)—other mechanisms, such as those based on digital signatures, provided evidence as to what information is authentically part of the audit trail.

Whether the audit trail is made public on a peer-to-peer based public ledger (as with bitcoin) or is made public on a website maintained by election officials is not the key question; the critical questions are whether the audit trail is readable by the public and whether there is reason to believe that it is complete and accurate (the sort of questions asked in a compliance audit).

The notion of having the audit trail totally public is a good one. It existed in the early days of our republic, but disappeared when secret ballots and voting machines became the norm. *It is time to again make election audit trails public.*

In the context of a public ledger containing the audit trail, a strongly software-independent voting system enables the reconstruction of the correct election outcome from the public audit trail.

[2]`http://www.coindesk.com/bitcoin-foundation-blockchain-voting-system-controversy/`

1.8 End-to-End Verifiable Voting Systems

“Cryptographic voting systems” were mentioned briefly in Section 1.3.2 as an example of (strongly) software-independent voting systems; here we elaborate a bit more on their properties, and their relationship to software independence and evidence-based elections.

This line of research has progressed significantly since our original paper on software independence was published (2006).

The common name for such systems has evolved to “*end-to-end auditable voting systems*” (or sometimes “*end-to-end verifiable voting systems*,” to emphasize that the verification covers all the way from the voter’s head (where her choices are selected) to the final outcome (reflecting all cast votes):

- a voter may verify that her vote is *cast as intended*,
- anyone may verify that a given vote is *collected as cast*, and
- anyone may verify that the votes are *counted as collected.*

In these systems, the collected cast votes are placed in a public ledger; to protect voter privacy, the votes are encrypted before being cast (e.g., with the public key of an election authority).

Some protocol, such as Benaloh’s “*ballot casting assurance*” protocol [85], is needed to assure voters that their votes are being properly encrypted. *Such a protocol is essential for making the design software independent*; without it the voting terminal could misrepresent the voter’s intent by encrypting something other than the voter’s choice.

For some designs, such as “Prêt à voter” [499], “Scratch and Vote” [49] and “Scantegrity” [386, 133, 146, 144], ballots are preprinted, containing both plaintext (human-readable) choices and corresponding ciphertexts. For such designs one should include a process (a “ballot audit”) for allowing voters (and other auditors) to randomly select preprinted ballots, spoil them (remove them from the pool of ballots eligible to be cast), and challenge the system to demonstrate that the ciphertexts properly represent the corresponding plaintexts. Again, such a ballot-auditing process is essential for making the design software independent; without it the ballot-printing subsystem could effectively cause voters’ selections to be represented incorrectly by the corresponding ciphertexts.

The “ThreeBallot” design of Rivest [487] was proposed primarily for pedagogic purposes to illustrate the principles of end-to-end verifiable voting system design *without using cryptography*. ThreeBallot was not intended as a practical proposal, since each voter must submit *three* ballots, which must have an enforced relationship to each other (vote exactly twice in favor of a candidate to support him, vote exactly once in favor of a candidate to oppose him). The voter retains a randomly chosen one of her three submitted ballots as a receipt so that she can check for its presence in the public ledger. The question as to whether ThreeBallot is software independent reduces to a consideration of the device that enforces the necessary relationship of the three submitted ballots. Clearly, the device should not know which of the three ballots was retained by the voter as her receipt.

However, a maliciously programmed device in ThreeBallot might allow a voter to submit *three* ballots in favor of a certain candidate, which should not be allowed. Is this a violation of software independence? It seems not, since it requires not only that the device software be changed, but also that some voters collude in submitting illegal triples of ballots. Perhaps a new definition is needed.

We may define a voting system to be *vote validating* if it checks that each cast vote is valid, and *publicly vote validating* if anyone may determine from the public ledger that each vote is valid.

We see that ThreeBallot is thus vote validating but not publicly vote validating.

Note that vote validation is not the same as providing ballot assurance; the former checks that the cast vote is one of the possible allowed votes, while the latter allows a voter to check that her cast vote correctly captures her intent.

Some end-to-end verifiable voting system designs, such as the homomorphic method proposed by Baudron et al. [76], achieve public vote validation by providing each vote with a zero-knowledge proof of its validity (the vote and corresponding zero-knowledge proof of validity are posted together on the public ledger).

It is worth noting that the inclusion of such zero-knowledge proofs may provide malicious software with a means to cause an election to fail to produce an output: what should happen when one (or many) of such zero-knowledge proofs of input validity fail to verify? This is outside the scope of the notion of software independence, since the activity of the malicious (or erroneous) software will of course be detected.

Some end-to-end verifiable voting designs, such as Scantegrity [386, 133, 146, 144] and STAR-Vote [77], are "paper/electronic hybrid" methods using a combination of paper-based and electronic methods, so that the paper ballot audit trail provides a backup mechanism for recovering the correct election outcome should the electronic or cryptographic methods completely fail somehow. This design also provides comfort to those who don't quite understand or trust the cryptographic techniques being used. Furthermore, the auditing process may include checks of both the paper audit trail and the electronic audit trail. (Should they disagree on the correct election outcome, the paper audit trail should probably take precedence, unless there is evidence that the paper audit trail was damaged or incomplete.)

Many proposed end-to-end verifiable voting system designs use mix-nets to provide voter privacy; the mix-net scrambles (permutes) the collection of cast votes while not adding or deleting votes, nor changing the content of any cast vote. To make a mix-net verifiable, the mix-net servers provide a zero-knowledge proof of these desired correctness properties; this zero-knowledge proof is also posted on the public ledger. In the absence of a paper audit trail, such zero-knowledge proof methods are essential for providing software independence.

The other major category of end-to-end verifiable voting system proposals are those that are based on encryption methods with homomorphic properties [49, 76, 77]. Such methods do not need zero-knowledge proofs of correct mix-net operation, since they do not use mix-nets; the homomorphic aggregation of votes provides the desired anonymity. However, achieving software independence requires some method (such as a zero-knowledge proof) that provides assurance that the decryption of the aggregated tally was correctly performed.

Recently we have seen increasing attention to the possibility of running elections remotely "over the internet." In particular, the question is asked as to whether end-to-end verifiable elections can be run over the internet.

While voting over the internet has been done in Estonia [563], Springall et al. [535] argue that the Estonian voting system is not end-to-end verifiable and that it has numerous security vulnerabilities.

The Helios voting system [44] is perhaps the most widely used internet-based end-to-end verifiable voting system. Like any remote voting system (such as vote-by-mail), there is no pretense of avoiding voter coercion; indeed, Helios makes the possibility of coercion explicit by providing a "Coerce-Me" button(!).

Küsters et al. [356] have demonstrated an interesting "clash" attack on some versions of Helios and on some other end-to-end voting systems, wherein voters who vote the same way may be given identical receipts (so when voters look them up on the public ledger everything seems OK, but the clash attack thereby provides the attacker with the freedom to add new ballots to the collection of cast votes). (The same authors also have an interesting definition of *accountability* applicable to voting systems [336].) Here the vulnerability lies with the random number generator; manipulating it can cause receipts to become identical. Systems with such a flaw are not software independent.

Remotegrity [586] is an interesting extension to the Scantegrity system, employing *both* paper and electronic communications to allow remote voters to detect whether their votes have been tampered with, and to prove that such tampering exists without having to reveal how they have voted. Although Remotegrity utilizes a complex protocol involving code voting and scratch-off cards mailed to the voters, it does appear to achieve software independence, among other properties.

1.9 Program Verification

As we discussed in Section 1.2.1, evaluating a software system for errors is generally held to be impossible. That said, approaches exist to verify that a software conforms to a given *specification*.

Given a specification S describing the input/output relationships, and a program P it is possible to write a formal proof π that interfaces of P satisfy the requirements outlined in S. Moreover such proofs π are verifiable. This is called *program verifi-*

cation and is an active research area. Thus, by spending a considerable and highly skilled effort it is, in principle, possible to produce a proof that software used in voting conforms to a specification.

However, for such proof to be relevant it must be possible to determine that the particular program is in fact being executed by the hardware, and that the hardware executes *nothing else* that could interfere with the said program. As demonstrated by Checkoway et al. [154] the latter is extremely hard and believed impossible in general.

1.10 Verifiable Computation and Zero-Knowledge Proofs

In early days of the field, program verification techniques comprised the only set of techniques to try proving output of a computation correct. A relatively recent approach, which circumvents the outlined impossibility outlined faced by program verification techniques, is based on zero-knowledge proofs. Here, an output of a program is augmented with a proof that the *output* conforms to the specification, and thus is correct for the given input.

This is consistent with the "evidence-based elections" theme described above (Section 1.6), following the mantra of "verify the outcome, not the equipment," and is the approach we examine further in this section.

Proofs for end-to-end voting systems, e.g., those that verify correct shuffling of a mixnet or that vote is well-formed, can be seen as tailored examples of such zero-knowledge proofs. In contrast, recent years have seen a spark of availability of efficient *general-purpose* zero-knowledge proof systems. Provided people trust and accept them, those could greatly expand the domain of cryptographically verified voting schemes.

In more detail, a zero-knowledge proof system, given a program P, input x and secret input w, produces the output $z := P(x,w)$ and a proof π attesting to the fact that $z = P(x,w)$. Anyone, given x, P, z and π, can be convinced that there exists w such that $z = P(x,w)$, however the proof reveals nothing about w other than its existence. A weaker variant called *verifiable computation* system, assumes that there is no secret input w.

Zero-knowledge proofs in voting. As we explain next, zero-knowledge proofs are very powerful cryptographic tools with immediate applicability to voting. Consider the scenario of counting encrypted votes. Here x could comprise encrypted votes, w be the decryption key held by the election officials, and P be the program that does the tallying. Any observer, given encrypted votes, final election result and the corresponding proof, can be convinced that votes were counted correctly. Moreover, the observer does *not* need to trust the hardware used to produce the proof, nor that P's computation was not interfered with, etc.

More generally, the beautiful line of zero-knowledge works [261, 347, 393, 251] have culminated in constructions that admit efficient practical prototypes [81, 454, 82]; we refer the reader to [258] for a survey. Most efficient constructions sport linear verification time and constant-sized proofs (in practice: verification in few milliseconds and proofs of a few hundreds of bytes, respectively). In particular, this efficiency means that most recent developments can greatly speed up existing voting primitives (e.g., verifiable mixnets) and support new ones (e.g., proofs of correct decryption for complex encryption schemes).

That said, from a cryptographic perspective, constructions of very efficient zero-knowledge proofs tend to be a bit "heavy-weight" — current proposals tend to require complex theoretical machinery or strong cryptographic assumptions.

Moreover, all non-interactive proof systems require a *trusted setup* phase which, if done improperly or maliciously, yields the proofs vacuous. This is in line with preparations for regular elections, where mistakes could potentially turn out to be fatal. However, there is recent theoretical work that tries to lessen the trust requirements of the setup phase, but the degree of practicality such a solution would provide remains to be evaluated.

1.11 Conclusions and Suggestions

The history of computing systems is that, given improvements and breakthroughs in technology and speed, software is able to do more and thus its complexity increases. The ability to prove the correctness of software diminishes rapidly as the software becomes more complex. It would effectively be impossible to adequately test future (and current) software-dependent voting systems for flaws and introduced fraud, and thus these systems would always remain suspect in their ability to provide secure and accurate elections.

A *software-independent* approach to voting systems assures voters that errors or fraud in election results can be reliably detected. Since the correctness of the election results does not ultimately depend on the correctness of the software, one can reduce the effort and expense to test and certify voting system software.