Auditing Self-Certifying Software is Trivial

Rajeev Goré

September 20, 2021

Abstract

We give a simple example of how to gain "software indpendence" via certificate producing software. We then recommend that the Commonwealth mandate that all Elections Commissions in Australia procure technology and software that is self-certifying.

Recommendations.

1. Require that all electronic voting and counting software produce public evidence which can be scrutinised by non-experts with ease.

Introduction. My name is Rajeev Goré and I was an ANU academic in Computer Science from 1994 until the end of 2020, and a professor there from 2011. My specialisation is in formal mathematical logic and I obtained my PhD in 1992 from the University of Cambridge, UK. I have served on the program committees of about 100 international conferences in this area, and most recently, I was a co-chair of the Technical Track of the International Conference on Electronic Voting and Identity (`https://www.e-vote-id.org/`). In March 2021, I was a keynote speaker at the International Conference on Formal Engineering Methods (`https://formal-analysis.com/icfem/2020/`). I am also on the editorial board of the International Journal on Logical Methods in Computer Science (`https://lmcs.episciences.org/page/editorial-board`). All of this is just to say "I know what I am talking about".

Sofware Independence via Self Certifying Software. Ron Rivest, a co-inventor of RSA public-key cryptography and a professor at MIT, suggests that all elections results should software-independent: *an undetected change or error in voting system software should be incapable of causing an undetectable change or error in an election outcome.* [3].

A Simple Example of Software Independence. Consider the following simple example of how we can obtain software independence.

Suppose the vendor's software has to accept two natural numbers x and y as **Inputs**, ensure that x is greater than or equal to y, and output the **Result** $x + y$. For example: with inputs 4 and 3, the output result should be 7; with inputs 0 and 0, the output result should be 0; and with inputs 3 and 4, the output should be *"error: 3 is not greater than or equal to 4."*.

For the sake of the example, pretend that addition is a very difficult task and requires thousands of lines of code so auditing the code is impossible.

We instead demand that the vendor's software outputs **Evidence** as below:

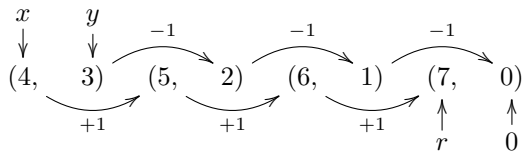
Evidence: the software must output as evidence a (possibly long) sequence of pairs of natural numbers which obey the following condition;

Checks: the first pair (x, y) is composed from the inputs x and y, the last pair $(r, 0)$ is composed from the claimed result r and the natural number 0, and every pair after the first simply increases the first component by 1 while decreasing the second component by 1 (without changing x and y).

So suppose the initial numbers are $x = 4$ and $y = 3$ which clearly satisfy that x is greater than or equal to y. Suppose that the software outputs the result $r = 7$ and outputs the evidence:

$$(4, 3) \quad (5, 2) \quad (6, 1) \quad (7, 0)$$

We can now check (trivially) that the first pair is equal to (x, y), that the first components $4, 5, 6, 7$ of the evidence do indeed increase by 1 while the second components $3, 2, 1, 0$ of the evidence always decrease by 1, until the final pair where the second component is 0 and the first component is the value of r (ie 7). We can even draw a diagram that visualises the moves to show the "+1" and "-1" conditions of the Check on this particular example:



Now suppose that, unbeknown to anyone, including the vendor, the software contains a bug or even multiple bugs, or is hacked by malicious actors.

Should we worry about the **Result** $4 + 3 = 7$ with the **Evidence** above?

No! Because we can prove two mathematical theorems about the natural numbers which together guarantee that if the output **Evidence** obeys the **Check** then the output **Result** is correct: that is, $r = x + y$ as required. The two theorems are these (whose proofs we can publish):

Theorem 1: for all natural numbers r, the sum of r and 0 is r; and

Theorem 2: for all natural numbers x and y, if x is greater than or equal to y and y is not 0 then $x + y$ is the same as $(x + 1) + (y - 1)$.

Theorem 1 is true by the definition of addition, and justifies the requirement that the last pair in the sequence must have a second component of 0. Theorem 2 just says that increasing the value of the first component by 1 while decreasing the value of the second component by 1 gives the same sum, thereby giving our **Check**. That is, the **Result** $r = x + y$ will be correct if the **Evidence** (sequence) passes the **Check**, even if the software contains bugs or is hacked.

Why should I trust your theorems? You shouldn't. You should hire your favourite mathematicians to check our published proofs.

But counting STV votes is much harder than simple addition. Yes it is, so the vital question is this:

> How to specify the form of **Evidence**, the form of the **Checks**, and prove the mathematical **Theorems** about STV vote-counting required to guarantee that the claimed result is correct with respect to the legislation if the evidence obeys the checks?

In 2019, we answered this question in the affirmative [1]. We also extended these notions to handle the many different types of STV elections used in Australia and derived formally verified code that can not only produce the required **Evidence** but that can also count up to 40 million votes in 20 minutes [2]. Indeed, we even produced formally verified software for checking the evidence! Thus STV counting software that produces evidence of the correctness of its result already exists. The vendor's just need to use it!

But why should we trust your evidence checker? You shouldn't. You should employ a good third year computer science student to write your own evidence checker according to the design instructions set out in our paper. The evidence checker is guaranteed to be easy to write, certainly much easier than auditing any actual counting software. Indeed, every political party should develop its own evidence checking software so that any anomalies in different implementations of the evidence checker can be ironed out and the election results are open to scrutiny by anyone.

Conclusion. There is no need for the auditor general to (somehow) audit complex vote-counting software. Just demand that the vendor publishes the form of evidence their software produces, the checks it must obey and the mathematical theorems and proofs that guarantee that if the evidence passes the checks then the result is correct. Any interested party, including the auditor general, can employ mathematicians to check the published proofs and employ a computer science graduate to write an evidence checker to ensure that the evidence obeys the checks (and hence that the result is correct).

References

[1] Milad K. Ghale. Verified and verifiable computation with STV algorithms, 2019.

[2] Milad K. Ghale, Rajeev Goré, and Dirk Pattinson. Modular synthesis of verified verifiers of computation with STV algorithms. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSEICSE 2019, 2019*, pages 85–94, 2019.

[3] Ronald 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.