OPINION ARTICLE

REVISED Rampant software errors may undermine scientific results [version 2; referees: 2 approved]

David A. W. Soergel[1,2]

[1]Department of Computer Science, University of Massachusetts Amherst, Amherst, USA
[2]Current address: Google, Inc., Mountain View, CA, USA

v2 **First published:** 11 Dec 2014, **3**:303 (doi: 10.12688/f1000research.5930.1)
Latest published: 29 Jul 2015, **3**:303 (doi: 10.12688/f1000research.5930.2)

Abstract
The opportunities for both subtle and profound errors in software and data management are boundless, yet they remain surprisingly underappreciated. Here I estimate that any reported scientific result could very well be wrong if data have passed through a computer, and that these errors may remain largely undetected. It is therefore necessary to greatly expand our efforts to validate scientific software and computed results.

Open Peer Review

Referee Status: ✓ ✓

	Invited Referees 1	2
REVISED version 2 published 29 Jul 2015	✓ report	✓ report
version 1 published 11 Dec 2014	? report	? report

1 **Titus Brown**, Michigan State University USA

2 **Daniel Katz**, University of Chicago USA

Discuss this article

Comments (1)

Corresponding author: David A. W. Soergel (david@davidsoergel.com)

How to cite this article: Soergel DAW. **Rampant software errors may undermine scientific results [version 2; referees: 2 approved]** *F1000Research* 2015, **3**:303 (doi: 10.12688/f1000research.5930.2)

Grant information: The author(s) declared that no grants were involved in supporting this work.

Competing interests: No competing interests were disclosed.

First published: 11 Dec 2014, **3**:303 (doi: 10.12688/f1000research.5930.1)

REVISED **Amendments from Version 1**

I agree with both reviewers that my claims were too strongly worded. I have softened the language throughout (including simply adding “may” to the title), and revised the abstract accordingly. I believe it is now clear that I am expressing a justifiable anxiety about computational errors affecting scientific results, but that I do not provide empirical evidence as to how often results really are invalid for this reason.

I added the entire section: “Popular software is not necessarily less bug-prone.”

In the conclusion, I clarified the relationship between correct results (our ultimate goal), software verification, and shared workflow systems.

See referee reports

Computational results are particularly prone to misplaced trust

Perhaps because of ingrained cultural beliefs about the infallibility of computation[1], people show a level of trust in computed outputs that is completely at odds with the reality that nearly zero provably error-free computer programs have ever been written[2,3].

It has been estimated that the industry average rate of programming errors is “about 15 – 50 errors per 1000 lines of delivered code”[4]. That estimate describes the work of professional software engineers-—not of the graduate students who write most scientific data analysis programs, usually without the benefit of training in software engineering and testing[5,6]. The recent increase in attention to such training is a welcome and essential development[7–11]. Nonetheless, even the most careful software engineering practices in industry rarely achieve an error rate better than 1 per 1000 lines. Since software programs commonly have many thousands of lines of code (Table 1), it follows that many defects remain in delivered code–even after all testing and debugging is complete.

Software errors and error-prone designs are compounded across levels of design abstraction. Defects occur not only in the top-level program being run but also in compilers, system libraries, and even firmware and hardware–and errors in such underlying components are extremely difficult to detect[12].

How frequently are published results wrong due to software bugs?

Of course, not every error in a program will affect the outcome of a specific analysis. For a simple single-purpose program, it is entirely possible that every line executes on every run. In general, however, the code path taken for a given run of a program executes only a subset of the lines in it, because there may be command-line options that enable or disable certain features, blocks of code that execute conditionally depending on the input data, etc. Furthermore, even if an erroneous line executes, it may not in fact manifest the error (i.e., it may give the correct output for some inputs but not others). Finally: many errors may cause a program to simply crash or to report an obviously implausible result, but we are really only concerned with errors that propagate downstream and are reported.

In combination, then, we can estimate the number of errors that actually affect the result of a single run of a program, as follows:

```
Number of errors per program execution =
    total lines of code (LOC)
    * proportion executed
    * probability of error per line
    * probability that the error
      meaningfully affects the result
    * probability that an erroneous result
      appears plausible to the scientist.
```

For these purposes, using a formula to compute a value in Excel counts as a “line of code”, and a spreadsheet as a whole counts as a “program”—so many scientists who may not consider themselves coders may still suffer from bugs[13].

All of these values may vary widely depending on the field and the source of the software. Consider the following two scenarios, in which the values are nothing more than educated guesses (informed, at least, by deep experience in software engineering).

Table 1. **Number of lines of code in typical classes of computer programs (via informationisbeautiful.net).**

Software Type	Lines of Code
Research code supporting a typical bioinformatics study, e.g. one graduate student-year.	O(1000) – O(10,000)
Core scientific software (e.g. Matlab and R, not including add-on libraries).	O(100,000)
Large scientific collaborations (e.g. LHC, Hubble, climate models).	O(1,000,000)
Major software infrastructure (e.g. the Linux kernel, MS Office, etc.).	O(10,000,000)

Scenario 1: A typical medium-scale bioinformatics analysis

- 100,000 total LOC (neglecting trusted components such as the Linux kernel).
- 20% executed
- 10 errors per 1000 lines
- 10% chance that a given error meaningfully changes the outcome
- 10% chance that a consequent erroneous result is plausible

Multiplying these, we expect that two errors changed the output of this program run, so **the probability of a wrong output is effectively 100%.** All bets are off regarding scientific conclusions drawn from such an analysis.

Scenario 2: A small focused analysis, rigorously executed

Let's imagine a more optimistic scenario, in which we write a simple, short program, and we go to great lengths to test and debug it. In such a case, any output that is produced is in fact more likely to be plausible, because bugs producing implausible outputs are more likely to have been eliminated in testing.

- 1000 total LOC
- 100% executed
- 1 error per 1000 lines
- 10% chance that a given error meaningfully changes the outcome
- 50% chance that a consequent erroneous result is plausible

Here **the probability of a wrong output is 5%.**

The factors going into the above estimates are rank speculation, and the conclusion varies widely depending on the guessed values. Measuring such values rigorously in different contexts would be valuable but also tremendously difficult. Nonetheless it is sobering that some plausible values can produce high total error rates, and that even conservative values suggest that an appreciable proportion of results may be erroneous due to software defects–above and beyond those that are erroneous for more widely appreciated reasons.

Put another way: publishing a computed result amounts to asserting that the likelihood of error is acceptably low, and thus that the various factors contributing to the total error rate are low. In the context of a specific program, the first three factors (# LOC, % executed, and errors/line) can be measured or estimated. However the last two ("meaningful change" and "plausible change") remain completely unknown in most cases. In the following two sections I argue that these two factors are likely large enough to have a real impact. It is therefore incumbent on scientists to validate computational procedures–just as they already validate laboratory reagents, devices, and procedures–in order to convince readers of the absence of serious bugs.

Software is exceptionally brittle

A response to concerns about software quality that I have heard frequently—-particularly from wet-lab biologists—-is that errors may occur but have little impact on the outcome. This may be because only a few data points are affected, or because values are altered by a small amount (so the error is "in the noise"). The above estimates account for this by including a term for "meaningful changes to the result". Nonetheless, in the context of physical experiments, it is tempting to believe that small errors tend to reduce *precision* but have less effect on *accuracy*–i.e. if the concentration of some reagent is a bit off then the results will also be just a bit off, but not completely unrelated to the correct result.

But software is different. We cannot apply our physical intuitions, because software is profoundly brittle: "small" bugs commonly have unbounded error propagation. A sign error, a missing semicolon, an off-by-one error in matching up two columns of data, etc. will render the results complete noise[16]. It is rare that a software bug would alter a small proportion of the data by a small amount. More likely, it systematically alters every data point, or occurs in some downstream aggregate step with effectively global consequences. **In general, software errors produce outcomes that are *inaccurate*, not merely *imprecise*.**

Many erroneous results are plausible

Bugs that produce program crashes or completely implausible results are more likely to be discovered during development, before a program becomes "delivered code" (the state of code on which the above errors-per-line estimates are based). Consequently, published scientific code often has the property that nearly every possible output is plausible. When the code is a black box, situations such as these may easily produce outputs that are simply accepted at face value:

- An indexing off-by-one error or other data management mistake associates the wrong pairs of X's and Y's[14,15].
- A correlation is found between two variables where in fact none exists, or vice versa.
- A sequence aligner reports the "best" match to a sequence in a genome, but actually provides a lower-scoring match.
- A protein structure produced from x-ray crystallography is wrong, but it still looks like a protein[16].
- A classifier reports that only 60% of the data points are classifiable, when in fact 90% of the points should have been classified (and worse, there is a bias in which points were classified, so those 60% are not representative).
- All measured values are multiplied by a constant factor, but remain within a reasonable range.

Software errors and statistical significance are orthogonal issues

A software error may produce a spurious result that appears significant, or may mask a significant result.

If the error occurs early in an analysis pipeline, then it may be considered a form of measurement error (i.e., if it systematically or randomly alters the values of individual measurements), and so may be taken into account by common statistical methods.

However: typically the computed portion of a study comes after data collection, so its contribution to wrongness may easily be independent of sample size, replication of earlier steps, and other techniques for improving significance. For instance, a software error may occur near the end of the pipeline, e.g. in the computation of a significance value or of other statistics, or in the preparation of summary tables and plots.

The diversity of the types and magnitudes of errors that may occur[17–21] makes it difficult to make a general statement about the effects of such errors on apparent significance. However it seems clear that, a substantial proportion of the time (based on the above scenarios, anywhere from 5% to 100%), a result is simply wrong—-rendering moot any claims about its significance.

Popular software is not necessarily less bug-prone

The dangers posed by bugs should be obvious to anyone working with niche or custom software, such as one-off scripts written by a graduate student for a specific project. Still it is tempting to think that "standard" software is less subject to these concerns: if everyone in a given scientific field uses a certain package and has done so for years, then surely it must be trustworthy by now, right? Sadly this is not the case.

In the open-source software community this view is known as "Linus's Law": "Given enough eyeballs, all bugs are shallow". The law may in fact hold when there are really many eyeballs reading and testing the code. However widespread *usage* of the code does not produce the same effect. This has been recently demonstrated by the discovery of major security flaws in two extremely widely used open-source programs: the "Shellshock" bug in the bash command line shell and the "Heartbleed" bug in the OpenSSL encryption library. In both cases, code that runs on a substantial fraction of the world's computers is maintained by a very small number of developers. Despite the code being open-source, "Linus's Law" did not take effect simply because not enough people read it–even over the course of 25 years, in the case of Shellshock.

This principle applies not only to the software itself, but also to computed results that are reused as static artifacts. For instance, it took 15 years for anyone to notice errors in the ubiquitous BLOSUM62 amino acid substitution matrix used for protein sequence alignment[22].

Furthermore, even popular software is updated over time, and is run in different environments that may affect its behavior. Consequently, even if a specific version of a package running on a specific computer is considered reliable, that trust cannot necessarily be extended to other versions of the same software, or to the software when run on a different CPU or on a different operating system[23].

What can be done?

All hope is not lost; we must simply take the opportunity to use technology to bring about a new era of collaborative, reproducible science[24–26]. Open availability of all data and source code used to produce scientific results is an incontestable foundation[27–31]. A culture of comprehensive code review (both within and between labs) can certainly help reduce the error rate, but is not a panacea. Beyond that, we must redouble our commitment to replicating and reproducing results, and in particular we must insist that a result can be trusted only when it has been observed on multiple occasions using **completely different software packages and methods.**

A flexible and open system for describing and sharing computational workflows[32] would allow researchers to more easily examine the provenance of computational results, and to determine whether results are robust to swapping purportedly equivalent implementations of computational steps. A shared workflow system may thereby facilitate distributed verification of individual software components. Projects such as Galaxy[33], Kepler[34], and Taverna[35] have made inroads towards this goal, but much more work is needed to provide widespread access to comprehensive provenance of computational results. Perhaps ironically, a shared workflow system must itself qualify as a "trusted component"–like the Linux kernel–in order to provide a neutral platform for comparisons, and so must be held to the very highest standards of software quality. Crucially, any shared workflow system must be widely used to be effective, and gaining adoption is more a sociological and economic problem than a technical one[36]. **The first step is for all scientists to recognize the urgent need to verify computational results–a goal which goes hand in hand with open availability of comprehensive provenance via workflow systems, and with verification of the individual components of those workflows.**

Competing interests

No competing interests were disclosed.

Grant information

The author(s) declared that no grants were involved in supporting this work.

Acknowledgements

Thanks to Annaliese Beery, Chris Warren, Eli Dart, and the reviewers C. Titus Brown and Daniel Katz for helpful comments on the manuscript.

References

1. Toby SB: **Myths about computers.** *SIGCAS Comput Soc.* 1975; **6**(4): 3–5.
Publisher Full Text

2. Bird J: **How many bugs do you have in your code?** *Java Code Geeks.* 2011.
Reference Source

3. Fishman C: **They write the right stuff**. fastcompany. 1996.
Reference Source

4. McConnell S: **Code complete**. Microsoft Press, Redmond, Wash. 2004.
Reference Source

5. Merali Z: **Computational science: Error, why scientific programming does not compute.** *Nature.* 2010; **467**(7317): 775–777.
PubMed Abstract | Publisher Full Text

6. Joppa LN, McInerny G, Harper R, *et al.*: **Computational science. Troubling trends in scientific software use.** *Science.* 2013; **340**(6134): 814–5.
PubMed Abstract | Publisher Full Text

7. Baxter SM, Day SW, Fetrow JS, *et al.*: **Scientific software development is not an oxymoron.** *PLoS Comput Biol.* 2006; **2**(9): e87.
PubMed Abstract | Publisher Full Text | Free Full Text

8. Seemann T: **Ten recommendations for creating usable bioinformatics command line software.** *Gigascience.* 2013; **2**(1): 15.
PubMed Abstract | Publisher Full Text | Free Full Text

9. Stodden V, Miguez S: **Best practices for computational science: Software infrastructure and environments for reproducible and extensible research.** *J Open Res Softw.* 2014; **2**(1): e21.
Publisher Full Text

10. Wilson G: **Software carpentry: Getting scientists to write better code by making them more productive.** *Comput Sci Eng.* 2006; **8**(6): 66–69.
Publisher Full Text

11. Wilson G, Aruliah DA, Brown CT, *et al.*: **Best practices for scientific computing.** *PLoS Biol.* 2014; **12**(1): e1001745.
PubMed Abstract | Publisher Full Text | Free Full Text

12. Thimbleby H: **Heedless programming: ignoring detectable error is a widespread hazard.** *Software: Practice and Experience.* 2012; **42**(11): 1393–1407.
Publisher Full Text

13. Zeeberg BR, Riss J, Kane DW, *et al.*: **Mistaken identifiers: gene name errors can be introduced inadvertently when using excel in bioinformatics.** *BMC Bioinformatics.* 2004; **5**: 80.
PubMed Abstract | Publisher Full Text | Free Full Text

14. Hall BG, Salipante SJ: **Retraction: Measures of clade confidence do not correlate with accuracy of phylogenetic trees.** *PLoS Comput Biol.* 2007; **3**(7): e158.
PubMed Abstract | Publisher Full Text | Free Full Text

15. Hutson S: **Data handling errors spur debate over clinical trial.** *Nat Med.* 2010; **16**(6): 618.
PubMed Abstract | Publisher Full Text

16. Chang G, Roth CB, Reyes CL, *et al.*: **Retraction.** *Science.* 2006; **314**(5807): 1875.
PubMed Abstract | Publisher Full Text

17. Beizer B: **Software testing techniques**. Van Nostrand Reinhold, New York, 1990.
Reference Source

18. Khannur A: **Structured Software Testing The Discipline of Discovering**. Partridge Pub. 2014.
Reference Source

19. Spinellis D: **Code Quality: The Open Source Perspective**. Adobe Press, 2006.
Reference Source

20. Vipindeep V, Jalote P: **List of common bugs and programming practices to avoid them**. Electronic, March, 2005.
Reference Source

21. Ray B, Posnett D, Filkov V, *et al.*: **A large scale study of programming languages and code quality in github**. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014.* New York, NY, USA, ACM, 2014; 155–165.
Publisher Full Text

22. Styczynski MP, Jensen KL, Rigoutsos I, *et al.*: **BLOSUM62 miscalculations improve search performance**. *Nat Biotechnol.* 2008; **26**(3): 274–275.
PubMed Abstract | Publisher Full Text

23. Gronenschild EH, Habets P, Jacobs HI, *et al.*: **The effects of FreeSurfer version, workstation type, and Macintosh operating system version on anatomical volume and cortical thickness measurements**. *PLoS One.* 2012; **7**(6): e38234.
PubMed Abstract | Publisher Full Text | Free Full Text

24. Hey T, Tansley S, Tolle K: **The fourth paradigm: data-intensive scientific discovery**. Microsoft Research, Redmond, Wash. 2009.
Reference Source

25. Mesirov JP: **Computer science. Accessible reproducible research.** *Science.* 2010; **327**(5964): 415–6.
PubMed Abstract | Publisher Full Text | Free Full Text

26. Nielsen MA: **Reinventing discovery: the new era of networked science**. Princeton University Press, Princeton, N.J. 2012.
Publisher Full Text

27. Barnes N: **Publish your computer code: it is good enough.** *Nature.* 2010; **467**(7317): 753.
PubMed Abstract | Publisher Full Text

28. Ince DC, Hatton L, Graham-Cumming J: **The case for open computer programs.** *Nature.* 2012; **482**(7386): 485–8.
PubMed Abstract | Publisher Full Text

29. Lees JM: **Open and free: Software and scientific reproducibility.** *Seismol Res Lett.* 2012; **83**(5): 751–752.
Publisher Full Text

30. Morin A, Urban J, Adams PD, *et al.*: **Research priorities. Shining light into black boxes.** *Science.* 2012; **336**(6078): 159–160.
PubMed Abstract | Publisher Full Text | Free Full Text

31. Sonnenburg S, Braun ML, Ong CS, *et al.*: **The need for open source software in machine learning.** *J Mach Learn Res.* 2007; **8**: 2443–2466.
Reference Source

32. Ludäscher B, Altintas I, Bowers S, *et al.*: **Scientific process automation and workflow management**. *Scientific Data Management: Challenges, Existing Technology, and Deployment, Computational Science Series*, 2009; 476–508.
Publisher Full Text

33. Goecks J, Nekrutenko A, Taylor J, *et al.*: **Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences.** *Genome Biol.* 2010; **11**(8): R86.
PubMed Abstract | Publisher Full Text | Free Full Text

34. Altintas I, Berkley C, Jaeger E, *et al.*: **Kepler: an extensible system for design and execution of scientific workflows**. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, IEEE, 2004; 423–424.
Publisher Full Text

35. De Roure D, Goble C: **Software design for empowering scientists.** *Software IEEE.* 2009; **26**(1): 88–95.
Publisher Full Text

36. Stodden VC: **The scientific method in practice: Reproducibility in the computational sciences**. 2010.
Publisher Full Text

Open Peer Review

Current Referee Status: ✓ ✓

Version 2

Referee Report 29 October 2015

doi:10.5256/f1000research.7365.r9698

✓ **Titus Brown**
Microbiology and Molecular Genetics, Michigan State University, East Lansing, MI, USA

This is a sober (and sobering) perspective on the likely frequency of software errors, and the author has addressed all of my concerns in the revision.

I have read this submission. I believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Competing Interests: No competing interests were disclosed.

Referee Report 04 August 2015

doi:10.5256/f1000research.7365.r9699

✓ **Daniel Katz**
Computation Institute, University of Chicago, Chicago, IL, USA

This version of the paper is much improved, and in general, I agree with the author's response comments. I still have some concerns with two issues, and reading the paper made me think of one more point of interest, but in general, I am satisfied to see this paper move forward as accepted.

The first issue I have is in the description of the scenarios, where the first says that in a somewhat typical analysis, the probability of a wrong output is 100%. This says to me that there is something wrong with the calculations, as I don't think typical outputs are 100% likely to be incorrect.

My second issue is the discussion of workflows in the final paragraph. While there are indeed a set of computations that are described as workflows, particularly data analysis applications, there is also a large set that are not workflows, particularly simulation applications, that are much better described as monolithic programs (though using subroutines, methods, objects, etc. internally). I don't think the workflow discussion is needed in this paper, or that it really helps make the key points.

Finally, as an additional point, it would be interesting to compare this paper with work done in fault tolerance (aka resilience) where errors in hardware lead to different types of errors in the application, ranging from hangs and crashes to detectable errors to insignificant or significant undetectable errors.

From my own work, http://www.slideshare.net/danielskatz/aft-dsk-reefinalreviewmay2001 provides an example of this, though there are likely many peer-reviewed published works that could also be used for a source of a comparison.

I have read this submission. I believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Competing Interests: No competing interests were disclosed.

Version 1

Referee Report 22 December 2014

doi:10.5256/f1000research.6338.r7096

Daniel Katz
Computation Institute, University of Chicago, Chicago, IL, USA

This opinion article makes a number of good qualitative points, and while I completely agree that there are errors in most software, I think the chances of those errors leading to incorrect published results are completely unknown, and could potentially be much smaller than the this paper claims. I think the basic claim in the title and the body of the paper may be dramatically overstated. The abstract says "most scientific results are probably wrong," but this itself seems wrong.

The author states, "we must insist that a result can be trusted only when it has been observed on multiple occasions using completely different software packages and methods."

First, I think this statement is overly focused on software. One method for developing trust in results from a particular code is that they match results from other codes. Another method is that they match results from experiment. A third method might be based on code review.

Second, this statement is not only true for software, it is also true for this complete paper. In order to believe the chances for errors claimed here, this paper itself needs to be verified, and not at the level of each assumption made internally (in the "How frequently ..." section), but at the level of the overall claim. This is not easy, but it would be worthwhile, similar to the author's statement, "Measuring such values rigorously in different contexts would be valuable but also tremendously difficult" (but at a different level). If "most scientific results are probably wrong," the author should be able to select a relatively small number of papers and demonstrate how software errors led to wrong results. I would like to see such an experiment, which would serve to verify this paper, rather than it standing as an unverified claim about verification.

Finally, there is the classic problem with verification of a model (software, in this case): that fact that it works well in one domain is no guarantee that it will work well in another domain.

Having made these objections to the degree of the illness of the patient, I mostly agree with remedies discussed in the last section. Open available of data and code is clearly good for both trust and reproducibility. Running (computational) experiments multiple ways can help finds any errors in any one of them, assuming they do not use common components (e.g., libraries, tools, or even hardware) that could lead to systematic biases. But how this should be done is less clear. For example, we have enough workflow systems that I don't see any need for any one of them to be more trusted than the code that runs on them; we can just use different workflow systems with different code as part of the testing.

Back to the author's last point, I agree that "to recognize the urgent need" is essential, but to me, the need is verification; I could read this closing comment as saying that the need is widely adopted and widely trusted workflow tools. This should be clarified.

In summary, this paper could be better titled and less strongly worded in places, and the paper itself needs to be verified. An alternate title would be one that makes the point, "Software, like other experiments, must be verified to be trusted"

I have read this submission. I believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.

Competing Interests: No competing interests were disclosed.

Author Response 20 Jul 2015

David Soergel,

Thanks very much for your insightful comments, and apologies for the long-delayed response. I believe I have addressed the main point about softening the claims throughout the paper. Some further thoughts follow:

"First, I think this statement is overly focused on software. One method for developing trust in results from a particular code is that they match results from other codes. Another method is that they match results from experiment. A third method might be based on code review."

I focus on software because I think it is commonly trusted far out of proportion with its level of validation. Everyone understands that physical measurements must be validated, devices must be calibrated, experiments should ideally be reproduced in other labs, etc.-- but code seems to be a cultural blind spot in this regard.

When a computational result can be directly compared to an experimental result, then of course agreement should increase trust in both. More commonly, I think, a given result arises from a combination of experiment ("data collection") and computation ("data analysis"), and comparisons can only be made between attempts incorporating both. Again agreement from multiple attempts should increase trust--but only if the analysis steps lack common components. This is another reason to focus on software: software is typically used downstream of data collection, so a bug can easily mask whatever signal is present in the underlying data, producing spurious agreement or spurious disagreement in the final result. Because software often has the last word in generating a result, then, it demands an even higher level of trust than upstream inputs.

Code review is certainly a good thing, but in my view is never sufficient to generate trust. Anecdotally, I've found plenty of bugs in code that was already reviewed. In any case, "code review" means many things to many people, and obviously the likelihood of finding bugs varies widely with the skill of the reviewer.

"Second, this statement is not only true for software, it is also true for this complete paper. In order to believe the chances for errors claimed here, this paper itself needs to be verified, and not at the level of each assumption made internally (in the "How frequently ..." section), but at the level of the overall claim. This is not easy, but it would be worthwhile, similar to the author's statement, "Measuring such values rigorously in different contexts would be valuable but also tremendously difficult" (but at a different level). If "most scientific results are probably wrong," the author should be able to select a relatively small number of papers and demonstrate how software errors led to wrong results. I would like to see such an experiment, which would serve to verify this paper, rather than it standing as an unverified claim about verification."

This is an opinion piece; I hope the more speculative language now makes clear that I am expressing justifiable anxiety that a serious problem may exist, rather than asserting that it definitely does exist. I certainly agree that verifying my estimates would be a great thing to do (particularly the aggregate error rate, not just the individual factors, as you point out). However I think that would be a major undertaking that is not tractable for me to do in this paper.

I do already cite a number of cases where software bugs resulted in wrong results, but these are basically anecdotal, and of course they are the ones that have already been found and reported. The proportion of these to the overall literature is vanishingly small. There are surely many more papers where an author or reader is privately aware of an error. And a still much larger proportion of papers, I believe (but cannot prove), contain errors that remain completely unknown.

I can think of only two ways to determine that proportion empirically. The first is to identify existing attempts to reproduce results, confirm that they are not subject to common sources of error, and track down the causes of any disagreement. This method may be subject to selection bias (i.e. in general, only important or controversial results get replication attempts in the first place).

The second is to take a random sample of papers and attempt to fully reproduce them, or at least to carefully review the code in search of errors. That would be really a lot of work-- in one example, an independent reproduction of a single computational study took 3 months. A systematic campaign to reproduce computational results would be great, inspired by similar efforts focusing on reproducing experimental results (e.g. the Amgen study and the OSF Reproducibility Project).

But I can't take it on alone! Rather I hope this paper helps to demonstrate the need for researchers, funders, and publishers to take code verification more seriously, and to foster the reproduction studies that would be needed to confirm or deny my estimates. Crucially, it's not just a matter of successfully running a study author's code (which, in the example case of in ACM conferences and journals, can be downloaded and compiled for only about half of the papers anyway). Journal policies requiring at least that level of replication would be a good start. But really the point here is to use different code to generate the same result.

So I think we agree: I am making an unverified claim about verification, and I too would like to see it verified.

"Finally, there is the classic problem with verification of a model (software, in this case): that fact that it works well in one domain is no guarantee that it will work well in another domain."

True, we can never make absolute guarantees. But we can do better than the status quo, which all too often provides no verification at all. Also, this point opens the question of the breadth of applicability of a given software artifact: some software does only a very specific thing, and so can be thoroughly verified within its single domain, while other software is very generic and so is much harder to verify across domains. I don't address this in the paper, except to the extent that the factor for "proportion of lines executed" is tangentially related (e.g., successful tests exercising some code paths say nothing about runs taking different code paths). That factor could be thought of more abstractly as the likelihood that verification in one domain should generate trust in another.

"Having made these objections to the degree of the illness of the patient, I mostly agree with remedies discussed in the last section. Open available of data and code is clearly good for both trust and reproducibility. Running (computational) experiments multiple ways can help finds any errors in any one of them, assuming they do not use common components (e.g., libraries, tools, or even hardware) that could lead to systematic biases. But how this should be done is less clear. For example, we have enough workflow systems that I don't see any need for any one of them to be more trusted than the code that runs on them; we can just use different workflow systems with different code as part of the testing."

I agree that the ideal way to gain trust in a particular result is to observe agreement from two experiments in which *everything* differs, even the workflow system and the hardware (at some length:
http://davidsoergel.com/posts/confirmation-depth-as-a-measure-of-reproducible-scientific-research
, but just see Fig. 2 there for the main point).

However: if we could agree on a common, trusted workflow system, that would make it much easier both to verify software components and to track down sources of error, simply by swapping out individual components of workflows with purportedly equivalent alternative implementations. When components are reused across workflows (and across labs, etc.), crowdsourced results from such component-swap experiments would quickly reveal which components are most commonly associated with robust results. I'll have to describe that vision more thoroughly elsewhere, but for now I hope it points at one thing I hope we could gain from standardizing workflow and provenance descriptions. Perhaps more simply: researchers are more likely to examine (and perhaps tweak and reuse) a workflow written in a language (or graphical notation, etc.) with which they are already familiar; Balkanization of workflow systems largely defeats their purpose.

"Back to the author's last point, I agree that "to recognize the urgent need" is essential, but to me, the need is verification; I could read this closing comment as saying that the need is widely adopted and widely trusted workflow tools. This should be clarified."

I did really mean both things--I've tried to clarify that.

Thanks again for the very helpful comments!

Competing Interests: No competing interests were disclosed.

Referee Report 16 December 2014

doi:10.5256/f1000research.6338.r7019

Titus Brown
Microbiology and Molecular Genetics, Michigan State University, East Lansing, MI, USA

David Soergel's opinion piece applies numerical calculations and common (software engineering) sense to thinking about errors in scientific software. I have seen no other piece that so simply and brutally summarizes the likely problems with current software development approaches in science, and I wholeheartedly agree with his recommendations. I think that the recommendation for a common trusted workflow system is an interesting one; I am particularly impressed by the point that we need separate implementations of important software, as this is often neglected by funding agencies and non-computational scientists.

The large majority of the points in the paper are well taken and should not be controversial except perhaps in aggregate!

The only major flaw in the paper is an overstatement of the central thesis. For example,
- The title is too definite; it needs a "probably" (which may decrease pithiness);
- Same with the abstract. One fix might be to eliminate the first sentence and move the last sentence to the top.
- For scenario 1, it's a pity there are no citations for these numbers, because they are nonintuitive (I found 20% executed to be too low, until I really thought it through, and then I agreed; but I'm not sure many people will believe). Is there any way to either bound or "suppose" these numbers a bit more?

I would say that if the statements can be softened a bit to indicate that all of this is *almost 100% certainly the case but we can't actually say it definitely* then the article would be very acceptable.

I didn't see the reference to the sign error debacle at the appropriate "sign error" point in the paper:
http://boscoh.com/protein/a-sign-a-flipped-structure-and-a-scientific-flameout-of-epic-proportions.html

Another reference that could be usefully added, given space:
http://www.fastcompany.com/28121/they-write-right-stuff

It might be worth adding a reference to the recent SSH debacle, where it turned out that incredibly well used software had a significant flaw. In other words, it's not enough for software to be well used for it to be correct! (Space permitting.)

I have read this submission. I believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.

Competing Interests: No competing interests were disclosed.

Author Response 18 Dec 2014
David Soergel,

Thanks for the kind and helpful comments! The editors prefer to wait for more reviews before issuing a revision, but in the meantime:
1. In the title, I could go with "...may undermine...". (The loss in pithiness is indeed a shame, but so be it).

2. Agreed re rearranging and toning down the abstract.

3. I'll do another search for references and justifications for the ballpark estimates of % LOC executed and so on, but I expect this one will be hard because there's so much variation. For now the numbers are just my intuitions based on experience; I can try to clarify that at least. I think the fuzziest one is the plausibility term-- I know of no effort to measure that, and am not even sure how you'd go about it. In the course of code development and data analysis, how often do you look at a result and think "that's just not right"? That one varies too with the paranoia level of the scientist. (e.g., I'm a big fan of doing sanity checks that may reveal that some result is not plausible, even if that fact was not immediately obvious).

Thanks for the citation suggestions. They're both already in there, actually (15 and 3, respectively), but I'll cite them from additional places as you suggest.

And yes, I'll add a para mentioning "Linus's Law" ("Given enough eyeballs, all bugs are shallow") and the recent counterexamples (Heartbleed, Goto Fail, and Shellshock). These are notable because they're all security vulnerabilities, which (perhaps rightly) get a lot more press than bugs of other kinds. There's also a world of difference between widespread *usage* and widespread *reading the code*-- a distinction that is sometimes glossed over in these discussions.

Thanks again for the comments!

Competing Interests: No competing interests were disclosed.

Author Response 20 Jul 2015

David Soergel,

Thanks again for the comments, and apologies for the long-delayed response. The changes described above are reflected in the new version.

Competing Interests: No competing interests were disclosed.

Discuss this Article

Version 1

Reader Comment 22 Dec 2014

Konrad Hinsen, Centre de Biophysique Moléculaire (CNRS), France

The problem discussed in this article is important indeed, and deserves experimental verification. The most obvious approch in my opinion is to have some computational method implemented twice, using tool chains as different as possible, and then compare the outcomes.

I have participated recently in two such experiments, for code of modest size but using a significant amount of infrastructure (compilers, libraries, ...). In both cases I wrote a Python implementation, using the Scientific Python ecosystem. In one case the other implementation was written in Matlab, in the other

Mathematica was used. Each of these implementations was written by a person with significant experience with his/her chosen platform.

For each problem, both authors tested their implementation until they considered it good for use in published work. Upon comparison, small differences were found and tracked down - fortunately they weren't just due to uncontrollable differences in floating-point computations. In both cases, both implementations turned out to have minor bugs. However, when the results ultimately agreed and more tests had been done, the final "official" results were not very different from what the original implementations had produced. The bugs were of the kind described in this article: an average computed over a data series minus the last point (off-by-one error), a wrong criterion in a data filter, a typo in a numerica constant, etc.

I think it would be interesting to do such studies on a much larger scale, and see if the article's estimates turn out to be reasonable.

Competing Interests: No competing interests were disclosed.