ARTICLE:

The Law Commission presumption concerning the dependability of computer evidence

An Invited Paper By

Peter Bernard Ladkin, Bev Littlewood, Harold Thimbleby and Martyn Thomas CBE

We consider the condition set out in section 69(1)(b) of the Police and Criminal Evidence Act 1984 (PACE 1984) that reliance on computer evidence should be subject to proof of its correctness, and compare it to the 1997 Law Commission recommendation that a common law presumption be used that a computer operated correctly unless there is explicit evidence to the contrary (LC Presumption). We understand the LC Presumption prevails in current legal proceedings. We demonstrate that neither section 69(1)(b) of PACE 1984 nor the LC presumption reflects the reality of general software-based system behaviour.

The Law Commission proposals

In Part XIII of *Evidence in Criminal Proceedings: Hearsay and Related Topics*,[1] the Law Commission considered in 1997 the rationale behind section 69 of the Police and Criminal Evidence Act 1984. Section 69 of PACE 1984 stated, among other things:

> (1) In any proceedings, a statement in a document produced by a computer shall not be admissible as evidence of any fact stated therein unless it is shown –
>
> > (a) that there are no reasonable grounds for believing that the statement is inaccurate because of improper use of the computer;
> >
> > (b) that at all material times the computer was operating properly, or if not, that any respect in which it was not operating properly or was out of operation was not such as to affect the production of the document or the accuracy of its contents; and ……..

Condition (1)(b) was considered by the Law Commission, accurately in our opinion, to be a significant imposition on those wishing to introduce evidence concerning computer operation, for two reasons.

First, for any moderately complex software-based computer system, such as the IT transaction-processing system Horizon used by Post Office Limited,[2] it is a practical impossibility to develop such a system so that the correctness of every software operation is provable to the relevant standard in legal proceedings. Any such proofs[3] require the use of mathematical-logical analysis methods (called formal methods) in the development of software. With the exception of certain computer-based OT[4] systems involved in safety-critical operations, such as aircraft control systems, at the time the Law Commission was writing (1997), the use of formal methods to guarantee correctness was not common in general IT-system software development. Nor is it so even today. Some of the authors have been involved for more than a decade in attempts to describe applicable, industrially mature formal methods in international standards for safety-critical OT systems. We can attest to the considerable resistance – even now – to any requirement for use of such methods, even in the development of such critical systems.

It is clearly impractical for a requirement such as set out in section 69(1)(b) of PACE 1984 to expect evidence of a type that can only be obtained by using methods which the software industry generally has not used, and remains resistant to using, and which

[1] The Law Commission, *Evidence in Criminal Proceedings: Hearsay and Related Topics* (1997), http://www.lawcom.gov.uk/app/uploads/2015/03/lc245_Legislating_the_Criminal_Code_Evidence_in_Criminal_Proceedings.pdf.

[2] Which has been the subject of recent legal proceedings, for which see *Bates v the Post Office Ltd (No 6: Horizon Issues)* [2019] EWHC 3408 (QB).

[3] It is important to note that attempts at such proofs might well not succeed, because the software is too complex for the proving technology. This is a standard difficulty with the application of such rigorous methods.

[4] "OT" stands for "operational technology", similar to "IT" for "information technology". Control systems run by digital computers are OT. The LC considers "documents produced by a computer", e.g., in paragraph 13.4. This looks to us as if it refers primarily to IT, since OT generally does not produce documents, but rather produces actions.

might in any given case not necessarily succeed very well. There may well be a reason for a law requiring that critical systems be developed using such methods, and such a law could then render a requirement such as that of PACE section 69(1)(b) feasible. But that law must come first, and it is not there yet.

Second, no matter the quality of the software, the computer hardware on which the software runs is necessarily constrained in the resources available to it. Hardware may, in the course of operations, be modified or constrained. By way of example, it may be exchanged or up-graded for other hardware, which may have different constraints on resources (such as size of memory or available disk space), or otherwise. These actions may cause the operations of even logically-impeccable software to act in such a way that they no longer fulfil their original intent; they may even behave unpredictably. This phenomenon is manifest in the increasing numbers of cybersecurity incidents, in which malware[5] is inserted into running systems to subvert their operations. Stuxnet (considered below) and Triton are examples of malware deliberately inserted into OT systems, in this case industrial process control systems, to cause them to fail (successfully in both cases). There is not even a theoretical technical solution to this drawback that will lead to reliable practical countermeasures.

The Law Commission wrote (original footnote omitted):

> 13.3 … section 69 … must be examined against the requirement that the use of computer evidence should not be unnecessarily impeded, while giving due weight to the fallibility of computers.
>
> 13.4 … section 69 … provides that a document produced by a computer may not be adduced as evidence of any fact stated in the document unless it is shown that the computer was properly operating and was not being improperly use …
>
> 13.5 In essence, the party relying on computer evidence must first prove that the computer is reliable… This can be proved by tendering a written certificate, or by calling oral evidence …

Here we note some technical terminology. In electrotechnical terms, "reliability" means "ability to perform as required, without failure, for a given time interval, under given conditions".[6] That is a notion of absolute perfection. However, as we shall note below, most software contains defects, at the rate (see our discussion below) of generally between 1 and 100 defects per 1,000 lines of source code (LOC; 1,000 LOC is often referred to as 1 kLOC).[7] The lower defect bound of around 1 per kLOC is generally attained only by carefully developed specialist safety-critical OT software, and not always. In general terms, none of us are aware of any non-trivial software-based system which can be shown to be reliable in the absolute sense given in the IEC definition.

There is a branch of software engineering that estimates failure rates of software in operation, and attempts to draw conclusions about the properties of the software from an analysis of the statistics. This branch is known as "software reliability" or "software reliability engineering".[8] Software reliability does not deal in perfection (for, as we remarked, we do not know of any practical instance of perfection in non-trivial software), but in estimating the chance of failure in operation, over a given time interval, to a given level of confidence (usually expressed either as a percentage or as a probability, equivalently). Such estimates are required, for example, by the UK Nuclear Regulator for assessing the performance of software-based emergency systems in UK nuclear power stations.

It is not yet clear to us from any legal reasoning we have read, largely about chances of failure of software in specific ways (e.g, *R v Seema Misra*,[9] and *Bates v Post Office Ltd (No 6: Horizon Issues)*[10]), how such

[5] That is, malicious software inserted into a system contrary to the intent of the manufacturer of the system. Malware is most often software, more rarely hardware, as in the cases mentioned. But cases in which hardware is surreptitiously reconfigured are not unknown.

[6] International Electrotechnical Vocabulary, definition 192-01-24, available at http://www.electropedia.org/iev/iev.nsf/display?openform&ievref=192-01-24.

[7] The average reported by Humphrey (see below) was more, but very carefully constructed code achieved less.

[8] The prestigious International Symposium on Software Reliability Engineering (ISSRE) has been running for over 30 years.

[9] T20090070, In the Crown Court at Guilford, Trial dates: 11, 12, 13, 14, 15, 18, 19, 20, 21 October and 11 November 2010, His Honour Judge N. A. Stewart and a jury, 12. *Digital Evidence and Electronic Signature Law Review* (2015) Introduction, 44 – 55; Documents Supplement.

[10] [2019] EWHC 3408 (QB).

general estimates would be of use in legal proceedings. Estimates that a *particular* failure phenomenon will or will not manifest over a certain period of use requires reviewing the data from many hours of operation (typically over many years – even many hundreds of years when covering a collection of systems) as well as meticulous recording of failure, to a degree of rigour that is not found in most IT operations. We comment further on the phenomenology of software failure and its statistical foundations in the Appendix.

The Law Commission cites Colin Tapper on IT-system error:

> 13.7 … As Professor Tapper has pointed out, "most computer error is either immediately detectable or results from error in the data entered into the machine".[13]
>
> 13 C Tapper, "Discovery in Modern Times: A Voyage around the Common Law World" (1991) 67 Chicago-Kent Law Review 217, 248.

Let us call the condition that a computer error "is immediately detectable or results from error in input data" the "Tapper Condition". We were surprised to read Tapper's suggestion that the Tapper Condition categorises "most computer error", even allowing that he was writing in 1991. Reading the original paper, it seems to us as if Professor Tapper was not categorising "most computer error" in unqualified terms, but rather considering particular phenomena that are manifest in the use of one specific sort of IT system, namely systems commonly used for clerical work (maybe, more specifically, for legal-clerical work). The Tapper Condition does not appear to hold in general.

For example, anyone who has used a spreadsheet program will be aware that many output errors result from incorrect calculations, and they are not immediately obvious.[11] Given the pervasiveness of arithmetical error in spreadsheet programs, explicated by Powell and others, the question arises how people in fact use these programs in practice, given that they are unreliable. It is beyond the scope of this paper to pursue this line of thinking further.

The "Pentium bug",[12] which was a fault in the design of the floating-point arithmetic processing in the Intel Pentium processor chip, manifestly did not satisfy the Tapper Condition. The bug was made public by Professor Nicely, a mathematics professor, a month after its maker knew of its existence. No one else using the CPU devices, of which there were very many, appears to have noticed inaccuracies in the output until Professor Nicely raised the issue.[13] Professor Nicely discovered the bug when he added a new Pentium processor to his set of computational devices performing number-theoretical research, and obtained some slightly different results when using the new processor. This was, in effect, a form of *regression test*. This is a pervasive and valued software assessment technique, in which, after an update to software, predefined tests which have been run on earlier versions of the software are repeated on the updated software to test whether exactly the same results ensue. Regression testing would be unnecessary if most bugs were overt and the Tapper Condition were to be valid.

Instances of what is called "unintended acceleration" were reported in certain models of Toyota car. There is a software-based defect which could have caused instances of unintended acceleration, which manifestly did not satisfy the Tapper Condition.[14] The existence of a bug which could cause the phenomenon was disputed for some years. It not only eluded discovery by the manufacturer, but also by a team of NASA specialists, who spent a number of person-years looking for one without success.[15] A bug was then discovered by the software engineer Michael Barr when working as an expert witness. He used a testing technique known as "fault injection". Barr claimed to have spent about three person-years discovering this bug. This demonstrates that such errors are not obvious – even to experts – and they do not satisfy the Tapper Condition.[16]

Our final example contradicting the Tapper Condition

[11] Stephen G. Powell, Kenneth R. Baker and Barry Lawson, 'Impact of errors in operational spreadsheets', *Decision Support Systems* 47(2):126-132, May 2009.

[12] This fault was in fact an error in hardware design, but came about in exactly the same way in which many software bugs come about – through mistakes in the design of the component.

[13] https://en.wikipedia.org/wiki/Pentium_FDIV_bug.

[14] Michael Barr, *Bookout v Toyota*, 2005 Camry L4 software analysis. Slides presented to the court, 2015, available at https://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf.

[15] Michael Barr, 'Firmware forensics: best practices in embedded software source code discovery', 8 *Digital Evidence and Electronic Signature Law Review* (2011) 148 – 151.

[16] We understand the manufacturer continues to dispute whether the actual phenomenon that was discovered was in fact responsible for the unintended acceleration cases.

is *R v Cahill; R v Pugh.*[17] Nurses were alleged to have falsified patient records in 2012, because of discrepancies found with computer records. The assumptions of the Tapper Condition led to a criminal trial, which collapsed three years later, when the cause of the discrepancies was admitted by a technician employed by the manufacturer of the IT system. Interestingly, this case revolved around a joint IT–OT system. The nurses used a handheld OT system, together with a complex back-office IT system recording their actions. That IT system was then corrupted, thus creating fallible evidence that had not been noticed was incorrect "immediately" in accordance with the Tapper Condition.[18]

Other renowned defects manifestly do not satisfy the Tapper Condition. For instance, malware represents a form of error, in that a system subject to the execution of malware will generally not perform its intended function. The Stuxnet malware succeeded in destroying a number of centrifuges before the phenomenon was discovered and operation of the equipment halted.[19]

In our opinion, inaccuracies in computer evidence are at least as likely to result from errors in the computer software as from errors in the data (including training data in the case of AI). Our examples above are intended to support this view, as well as show the limited applicability of the Tapper Condition.

After noting various problems with a rigorous application of PACE 1984 section 69 in the light of the way in which IT systems were developed and monitored, as well as that some OT systems (for instance, an intoximeter) allow alternative methods of verifying the accuracy of their operation, the Law Commission proposed:

> 13.13 … that section 69 of PACE be repealed without replacement.[25] Without section 69, a common law presumption comes into play:
>
> > In the absence of evidence to the contrary, the courts will presume that mechanical instruments were in order at the material time.[26]
>
> 25 Ibid.
>
> 26 Phipson, para 23-14, approved by the Divisional Court in *Castle v Cross* [1984] 1 WLR 1372, 1377B, *per* Stephen Brown LJ.
>
> 13.14 Where a party sought to rely on the presumption, it would not need to lead to evidence that the computer was working properly on the occasion in question unless there was evidence that it may not have been – in which case the party would have to prove that it was (beyond reasonable doubt in the case of the prosecution, and on the balance of probabilities in the case of the defence). The principle has been applied to such devices as speedometers[27] and traffic lights,[28] and in the consultation paper we saw no reason why it should not apply to computers.
>
> 27 Nicholas v Penny [1950] 2 KB 466.
>
> 28 *Tingle Jacobs & Co v Kennedy* [1964] 1 WLR 638n.

There are, in fact, significant differences between various digital computer-based systems that render such a pervasive assumption questionable. The Law Commission cites relatively uncomplex OT systems such as speedometers, traffic lights, and intoximeters. Such systems can indeed mostly satisfy the Tapper Condition of overt, immediately recognisable failure, or indeed recognisable failure upon inspection, in contrast to the examples we have cited above that manifestly do not.

We note that OT is manifestly not "computer evidence" as such, although system logs may constitute such evidence. Evidence is presumed to be information, and is therefore overtly IT. However, we note that the Law Commission introduces examples from OT when formulating its views on IT, including its view on the Tapper Condition – it mentions an intoximeter, traffic lights and speedometers, which are all OT. We consider these examples below.

To understand how an intoximeter might satisfy the Tapper Condition, we note that there are physically independent ways to verify the results of an intoximeter application. A laboratory test for blood-alcohol level is available; the subject may exhibit other, independent evidence of intoxication (some police forces ask a subject to walk along a narrow straight line); and so on.

A second OT example is traffic lights. Traffic lights

[17] 14 October 2014 (Crown Court at Cardiff, T20141094 and T20141061 before HHJ Crowther QC).
[18] Harold Thimbleby, "Misunderstanding IT: Hospital cybersecurity and IT problems reach the courts," *Digital Evidence and Electronic Signature Law Review*, 15:11–32, 2018. DOI 10.14296/deeslr.v15i0.4891; Ruling by the Judge, 14 *Digital Evidence and Electronic Signature Law Review* (2017) 67 – 71. DOI 10.14296/deeslr.v14i0.2541
[19] Ralph Langner, *To Kill a Centrifuge*, The Langner Group, November 2013, available from https://www.langner.com/to-kill-a-centrifuge.

appear to be fairly simple in terms of basic operation, but functioning devices certified for operation on roadways in most developed countries have complex and sophisticated failure-tolerance mechanisms built into them. These safety properties (such as not showing "proceed" simultaneously in two conflicting directions of travel) are implemented on the basis of previous (often unfortunate) experience. The failure-tolerant mechanisms introduced for this purpose are generally validated by independent inspection agencies. Such rigour is not generally applied to, nor can it likely be practically applied in the same way to, such large IT systems as, say, the Post Office Horizon system. It is (still) an open question as to what software-based systems might rigorously satisfy the Tapper Condition.

The Law Commission continues by discussing the response to their consultation proposal. It concludes:

> Our recommendation
>
> 13.23 We are satisfied that section 69 serves no useful purpose. We are not aware of any difficulties encountered in those jurisdictions that have no equivalent. We are satisfied that the presumption of proper functioning would apply to computers, thus throwing an evidential burden on to the opposing party, but that that burden would be interpreted in such a way as to ensure that the presumption did not result in a conviction merely because the defence had failed to adduce evidence of malfunction which it was in no position to adduce. We believe, as did the vast majority of our respondents, that such a regime would work fairly. We recommend the repeal of section 69 of PACE.[45] (Recommendation 50)
>
> 45 See cl 19 of the draft Bill.

Section 69 of PACE 1984 was repealed by section 60 of the Youth Justice and Criminal Evidence Act 1999. Since then, the presumption mentioned by the Law Commission in its recommendation 13.23 (what we have called the LC Presumption) has prevailed as a rule of evidence.

Our discussion above indicates that we do not consider that the LC Presumption appropriately considers the actual behaviour of all software-based systems, at least in the current or foreseeable state of engineering development. We have indicated that certain superficially-simple OT systems based on digital computer hardware and software, such as traffic lights, speedometers and intoximeters may satisfy the Tapper Condition and therefore justify the LC Presumption. However, such systems are usually based on validation and certification regimes that are completely different from the current techniques used in the development of large software-based IT systems, such as the Post Office Horizon system. Even much simpler IT systems are remarkably hard to verify and validate.

It is well to consider the quality of software in general. The facts are, in our opinion, not encouraging. We now review some of them.

The quality of software

The quality of software is traditionally taken in software engineering to be correlated with the density of (discovered) defects.[20]

First, we provide some definitions. A defect is, according to the influential definition used at Carnegie Mellon's Software Engineering Institute, "any flaw or imperfection in a software work product or software process", in which by *software work product* is meant any artefact created as part of the *software process*, which itself is "a set of activities, methods, practices, and transformations that people use to develop and maintain software work products".[21]

The reader will recall we explained 1 kLOC as 1,000 lines of code; here we need to be a little more precise. In the numbers we cite below, we consider lines of executable source code, 1 kLOC represents a very small program. The term "executable" means that lines of pure commentary are not included in the count. Commentary occurs regularly in well-written source code as a means of leading an inspector through its intended operation. "Source code" here generally means programs written in imperative languages such as C, Ada, Java, and Python. Such imperative languages are by far the most common type of language used for programming IT systems. Such source code is transformed by automated formal linguistic processes (compiling, and linking) into code which can be read and executed directly by a

[20] Yet another term for "something wrong". We introduce it here because it is used by the literature we discuss immediately below.

[21] Brad Clarke and Dave Zubrow, *How Good is the Software: A Review of Defect Prediction Techniques*, Carnegie-Mellon University Software Engineering Institute presentation, 2001, available from https://resources.sei.cmu.edu/asset_files/Presentation/2002_017_001_22912.pdf.

computer.

Defects can arise in a number of ways in this process of writing source code in a higher-level language and translating. They may occur in the source code itself, or be introduced by the largely-automated processes of implementation, and lastly by the computer hardware not quite executing the machine code the way its designers intended or expected. Examples of all of these phenomena are known to well-informed software engineers. We consider here exclusively defects in source code.

We now consider what defect numbers look like in terms of kLOC. For the sake of simplicity, we only consider defects in source code, ignoring other sources of errors.

Humphrey considered data derived from more than 8,000 programs written by industrial software developers.[22] He wrote, "We now know how many defects experienced software developers inject. On average, they inject a defect about every ten lines of code." The average number of defects per kLOC was about 120. The best 20% of programmers managed 62 defects per kLOC; the best 20%, 29 defects per kLOC. Even the top 1% still injected 11 defects per kLOC.[23] Typical OT and IT software has many kLOCs, even thousands of kLOCs, and hence very many defects. The evidence implies that all software can be considered to have multiple faults.

McDermid and Kelly reported on the defect densities in safety-critical industrial software:[24]

> "There is a general consensus in some areas of the safety critical systems community that a fault density of about 1 per kLoC is world class. Some software … is rather better but fault densities of lower than 0.1 per kLoC are exceptional. The UK [Ministry of Defence] funded the retrospective static analysis of the [Hercules] C130J [transport aircraft] software, previously developed to [civilian aerospace software standard RTCA] DO-178B, and determined that it contained about 1.4 safety-critical faults per kLoC (the overall flaw density was around 23 per kLoC…whilst a fault density of 1 per kLoC may seem high it is worth noting that commercial software is around 30 faults per kLoC, with initial fault injection rates of over 100 per kLoC."

Consider that "safety-critical faults" means faults whose possible consequences include system failures causing damage (injuries or death and/or damage to the environment). German and Mooney[25] and German[26] report the C130J static analysis to which McDermid and Kelly refer. Also see the somewhat different view of Daniels.[27] Ladkin lists the specific software defects identified during this static analysis, which were communicated to one of the authors.[28] Jackson, Thomas and Millett reference a plethora of incidents and dependability problems.[29]

Such defect densities are not inevitable. For example, a company founded by one of the authors, Praxis, now part of Altran UK Limited, develops software using a program development environment named SPARK,[30] which makes extensive use of formal methods. Chapman reports general defect densities of around 1 per 25 kLOC in delivered software developed

[22] Watts S. Humphrey, 'The *Watts New?* Collection: Columns by the SEI's Watts Humphrey', Special Report CMU/SEI-2009-SR-024. Software Engineering Institute, Carnegie-Mellon University, November 2009, available at https://resources.sei.cmu.edu/asset_files/SpecialReport/2009_003_001_15035.pdf.

[23] 'The *Watts New?* Collection: Columns by the SEI's Watts Humphrey', p 132.

[24] John McDermid and Tim Kelly, *Software in Safety-Critical Systems: Achievement and Prediction*, Nuclear Future 02(03), 2006, 3.1. Preliminary version is available at https://www-users.cs.york.ac.uk/tpk/inuce2004.pdf.

[25] Andy German and Gavin Mooney, 2001. Air Vehicle Software Static Code Analysis – Lessons Learnt, in Felix Redmill and Tom Anderson (eds.), Aspects of Safety Management, Proceedings of the Ninth Safety-critical Systems Symposium, Springer-Verlag London.

[26] German, Andy 2003. Software Static Code Analysis Lessons Learned. Crosstalk 16(11), The Journal of Defence Software Engineering, November 2003, available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.638.6456&rep=rep1&type=pdf.

[27] Daniels, Dewi, 2019 Email Contributions concerning the UK MoD C130J software inspection, sent to the System Safety List, 2019-11-14 and 2019-11-17, available at http://www.systemsafetylist.org/2019-November/005033.html, http://www.systemsafetylist.org/2019-November/005044.html and http://www.systemsafetylist.org/2019-November/005045.html

[28] Peter Bernard Ladkin, 2011. Dependable Software: A View. Slides for a keynote talk at the 2011 Ada Connection conference, Edinburgh UK. Available from https://rvs-bi.de/publications/Talks/AdaConn2011TalkLadkin.pdf.

[29] Daniel Jackson, Martyn Thomas and Lynette I. Millett (eds.), 2007. *Software for Dependable Systems: Sufficient Evidence? Report of the Committee on Certifiably Dependent Software Systems*, (U.S.) National Research Council. National Academies Press. Available from https://www.nap.edu/catalog/11923/software-for-dependable-systems-sufficient-evidence#toc.

[30] John Barnes, *SPARK: The proven approach to high-integrity software* (Altran Praxis, 2012).

using SPARK. This is one to two orders of magnitude better than the C130J software considered above.[31]

However, as we have noted above, many software developers still eschew thorough use of such formal methods as used at Praxis and Altran UK Limited. In our view this is because the use of formal methods requires a level of mathematical and logical understanding that many professional programmers typically still do not possess. In addition, training and experience in the use of such methods constitutes a cost that most companies are not necessarily willing to pay unless they are required to do so by contractual or regulatory obligation.

There are exceptions. The current international standard for the functional safety of software is IEC 61508-3:2010.[32] This standard, as well as its predecessor from 1997, classifies "formal methods" as "Highly Recommended" for the highest-dependability categories of software-implemented safety function (those needing a "Systematic Capability" SC4, and sometimes also for the next-lower category SC3). It does not amplify, however, on which methods might be meant, or which use of them would be helpful or appropriate.

In an area of rapidly increasing importance for software, the quality of software is generally taken by cybersecurity professionals to be connected to its cybersecurity vulnerability. This is shown in, for example, the training materials of the SANS Institute.[33] We note that the list of vulnerabilities in Tables C1 to C7 of the NIST guidance on ICS cybersecurity[34] look remarkably like lists of software defects. While this may be so, we note that not every software defect results in a cybersecurity vulnerability, and not every cybersecurity vulnerability arises from a software defect.

A "Third Way" between PACE 1984 and the LC Presumption

Computers are indeed fallible, as the Law Commission recognised (paragraph 13.3). Any IT system of practical size will have displayed faults many times since it was first put into service. Most IT systems will have had very many faults corrected through "patches" or new releases of software. No programmer could credibly claim that they know that they have corrected the last fault in their software.[35] We suggest that no competent programmer would even make such a claim.

From the discussion above, it is our view that a court should start with the presumption that any software system contains or is influenced by errors that make it fallible. It will therefore fail from time to time when a combination of circumstances lead to an erroneous path of execution through the software – and such failures may not be obvious, and may even be perverse. In assessing the weight to be placed on specific computer evidence, it follows from this that the trier of fact should ask "how likely is it that this particular evidence has been affected in a material way by computer error?"

Providing an answer to this question involves, first, reviewing any available evidence for the number, frequency and nature of errors that have been reported in the particular system previously.

Relevant evidence should normally be readily available. Any professional software support team will of necessity maintain a database or log of errors that have been reported, investigated and perhaps corrected. There is likely to be a "known error log" and records of earlier fixes. There may be lists of corrected errors, and lists of errors that remain to be corrected. These will be summarized in release notices that accompany technical fixes ("patches") or new releases of software components. Further details may exist in project reports and email exchanges. If the software team works to professional project-management standards, these should define what records are kept and what information is available, and how it has been audited. We note that, with the

[31] Roderick Chapman, Slides 43-48 of presentation Correctness by Construction: The Case for Constructive Static Verification, 2005, available at. https://samate.nist.gov/SSATTM_Content/papers/Correctness%20by%20Construction%20-%20Chapman.pdf.

[32] International Electrotechnical Commission, IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements, Edition 2, 2010.

[33] By way of example, see Evelyn Labbate, *Vulnerability as a Function of Software Quality*. Global Information Assurance Certification Paper, version of 2018-03-18. SANS Institute. Available at https://www.giac.org/paper/gsec/647/vulnerability-function-software-quality/101493.

[34] Keith Stouffer, Suzanne Lightman, Victoria Pillitteri, Marshall Abrams and Adam Hahn, 2015. Special Publication 800-82**,** Guide to Industrial Control Systems (ICS) Security, Revision 2. U.S. National Institute of Standards and Technology, May 2015. Available from https://csrc.nist.gov/publications/detail/sp/800-82/rev-2/final.

[35] See, for example, the discussion of the work of Adams and others in the Appendix.

rise in concerns about cybersecurity, such record-keeping is mandated by IT security standards. Maintaining logs of system events, for example, is part of an Information Security Management System (ISMS), as required by ISO/IEC 27001:2013[36] (Annex A 12.4).

In *Bates v Post Office Ltd (No 6: Horizon Issues)*,[37] the company Fujitsu, which was responsible for the IT system Horizon, maintained a Known Error Log (KEL), and maintained records of reported potential anomalies and follow-up action, including remedial measures by way of records known as PEAKs.[38] This enabled Fraser J and the expert witnesses to categorise known Horizon system defects. As a result, Fraser J was able to arrive at a judgement concerning whether the claimants might be right that anomalies in the Horizon system, and not their own malfeasance, caused the problems they were accused of by the Post Office.

Second, we suggest a court should consider the nature of the specific evidence. Specifically, could the evidence be materially changed by computer error?

For example, suppose evidence were to be presented in the form of a coherent and lengthy chain of emails nominally between participants. It would be hard to propose a credible computer error that could create such a lengthy email chain of any complexity that retained coherence. In contrast, consider a single calculated numerical value in a spreadsheet adduced as evidence. It is easy to see how such a calculated numeric value could be incorrect because of an erroneous algorithm underlying the spreadsheet calculations; indeed, many such errors are commonly encountered.[39]

Third, there is the question whether an IT system complies (and, if so, to what level of conformance) with any relevant standards to the application that it nominally serves. For instance, consider the bookkeeping system in relation to *Bates v Post Office Ltd (No 6: Horizon Issues)*.[40] It is a requirement for commercial bookkeeping to record all transactions. Its is also important not to record transactions that do not occur. It would follow that an IT system that performs commercial bookkeeping functions should adhere to these requirements, amongst others. However, it seems that Horizon did not: Fraser J noted a phenomenon listed as “Phantom Transactions”. (See number 15 in his list of 29 bugs exhibited by Horizon and known to Fujitsu in Appendix 2 to his judgment.)

An IT system which serves a commercial application, but which does not adhere to standard requirements within that application, can be regarded as generally less dependable than systems which do so adhere, and a court can legitimately make such an inference. An exception to this inference may occur when the IT system is accompanied by appropriate documentation and evidence that explicitly lists the application requirements the system does and does not address; and that this documentation is in fact correct. It is presumed that the users are aware of this documentation, its contents and implications.

Implications for the LC Presumption

We have considered the requirement of PACE 1984 section 69, the 1997 Law Commission review of computer evidence which considered the repeal of the section, the deliberations of that review, including the Tapper Condition and various phenomena of IT and OT, including the discussion of a number of relevant examples. We conclude that neither section 69(i)(b) of PACE 1984 nor the LC Presumption reflects the reality of general software-based system behaviour.

In the Appendix, we provide a technical account of the mathematical and statistical models, together with some empirical and experimental results that support the current understanding of software failure processes within the software engineering community. In particular, we consider how this

[36] International Organization for Standardization/International Electrotechnical Commission, ISO/IEC 27001 Information technology -- Security techniques -- Information security management systems – Requirements, ISO and IEC, 2013.
[37] [2019] EWHC 3408 (QB).
[38] Fraser J at [621] ‘The experts agreed the following about PEAKs and their content. “PEAKs record a timeline of activities to fix a bug or a problem. They sometimes contain information not found in KELs about specific impact on branches or root causes – what needs to be fixed. They are written, by people who know Horizon very well. They do not contain design detail for any change. They are generally about development activities and timeline rather than about potential impact. PEAKs typically stop when development has done its job, so they are not likely to contain information about follow-on activities, such as compensating branches for any losses.” It is also agreed, and indeed can be seen from the actual PEAKs themselves, that some of them record observations of financial impact.’
[39] Stephen G. Powell, Kenneth R. Baker and Barry Lawson, ‘Impact of errors in operational spreadsheets’, *Decision Support Systems* 47(2):126-132, May 2009.
[40] [2019] EWHC 3408 (QB).

understanding allows us to reach a number of conclusions about the inappropriateness of the statement by the Law Commission at 13.23: "We are satisfied that the assumption of proper functioning would apply to computers…"

This statement seems to us to come close to asserting that it should be assumed that a software fault is not the cause when some untoward event has occurred, unless there is overt evidence of such a fault. Ironically, a poorly engineered system, which is likely to suffer from the effects of bugs, may also be unlikely to record reliable evidence of its own behaviour.

In the particular case of evidence about the reliability of IT and non-trivial OT, there are three issues, based on the current state of knowledge of software engineering, upon which we regard it as necessary that a court form a view when considering computer evidence. The points noted below are more fully elaborated upon in the Appendix:

> (1) A presumption that any particular computer system failure is not caused by software is not justified, even for software that has previously been shown to be very reliable.
>
> (2) Evidence of previous computer failure undermines a presumption of current proper functioning.
>
> (3) The fact that a class of failures has not happened before is not a reason for assuming it cannot occur.

Peter Bernard Ladkin - Bielefeld University. CEO of tech-transfer companies Causalis Limited and Causalis Ingenieurgesellschaft mbH.

Bev Littlewood - Emeritus Professor of Software Engineering, Centre for Software Reliability, City, University of London.

Harold Thimbleby - Emeritus Professor, Gresham College, London; See Change Digital Health Fellow, Swansea University, Wales; Visiting Professor, UCL, London.

Martyn Thomas CBE - Emeritus Professor, Gresham College, London; Visiting Professor of Software Engineering at Aberystwyth University, Wales.

The authors have collectively 150 years experience in research and practice of software engineering, in particular with mathematical and logical methods for dependability, much of it safety-related. Their practical methods for incident analysis are used by 11,000 engineers world-wide, and for dependable-software/system development by companies in the UK, France and the US. They have founded four engineering companies, won the IEEE Harlan D. Mills Award for software engineering, been honoured by the Queen for services to software engineering, and have advised the judiciary, lawyers, regulators and government, and been a regulator.

The authors have also, formally and informally, acted as reviewers to chapter 6 'The presumption that computers are 'reliable'' in Stephen Mason and Daniel Seng, editors, *Electronic Evidence* (4th edn, Institute of Advanced Legal Studies for the SAS Humanities Digital Library, School of Advanced Study, University of London, 2017), Open Access PDF version in the Humanities Digital Library http://ials.sas.ac.uk/digital/humanities-digital-library/observing-law-ials-open-book-service-law/electronic-evidence

Appendix

A general account of the software failure process, and its consequences for the LC Presumption

In this appendix we provide support for our three concluding bullet points (for which see below) concerning the LC Presumption. We do this by first discussing the nature of software faults and failures. We then describe two widely accepted mathematical models of the software failure process. We go on to describe briefly some quantitative evidence about software faults and failures, obtained from experience of widespread operational experience of large software systems, and from some innovative experiments. We then use this chain of model and evidence to support our critique of the LC Presumption.

We begin with some terminology. A *software fault*[41] is something "not right" in the software. It is the result of some erroneous action during the creation of the software, or perhaps at some later stage, for instance when an attempt to fix a fault results in the introduction of a new one. Such erroneous action can be human error, for example a programmer error, or, when software is automatically generated from higher-level functional descriptions, as is increasingly common, an error in the generation process.

Software in the form of source code can be regarded as "pure design" since it has no physical manifestation, unlike hardware. Software faults are the result of errors in design. Software faults are static: they are a permanent characteristic of software until they are corrected. Unlike hardware, software does not "break"; it does not suddenly start behaving differently without anything else having changed, as a chip does if transistors burn out. Note that software faults can be either errors of commission – something that is done that is "wrong" – or errors of omission – something not done that should have been done.

A software *failure* is an event in which the software does not exhibit the expected or intended behaviour or yield the expected or intended output. Suppose the software under consideration has a fault. Then, given certain input data to the software in a particular operational environment, the fault may become manifest, in that it causes a failure of the software. Suppose a failure has occurred and has been observed. There usually follows a search for the fault that caused it. The fault can be considered initially as the sum total of characteristics of the software that contributed to causing the failure. Not all of those characteristics will necessarily be erroneous; some of them might well be characteristics one wants to retain. But some feature in that sum total will have to be changed if we do not want the failure to recur. Usually, the feature or features chosen to be changed are regarded as the kernel "fault".

There are two usual models – mathematized conceptions – of software operational failures: one for discrete events and a second for continuous operation. An example of a discrete-demand system is the protection system of a nuclear reactor, which is called upon to shut down the reactor and keep it safe if the reactor gets into a hazardous state. Such a system is only called upon to act, generally rather infrequently, when a hazardous state is entered. That call is known as a *demand*, an event taken to occur at a discrete point in time. Failure to act upon such a demand is a potentially serious event for a safety-critical system, such as a reactor protection system. Many commercial IT systems are also demand-based, for example, the transaction-processing part of the Horizon system, considered in *Bates v Post Office Ltd (No 6: Horizon Issues)*,[42] *R v Seema Misra*[43] and the earlier case of *Post Office Ltd v Castleton*.[44] Here, an interaction of a subpostmaster with the system is a demand, and each interaction can be correctly handled by the system, or the system can fail in some way. If we are looking just at failure behaviour, we are not concerned with the mathematical details of the transaction, but only in whether the outcome was *success* (a successful transaction) or *failure* (something went wrong).

Examples of continuously operating systems are common in engineering when a physical system is under computer control. An example is a fly-by-wire

[41] "Fault" is also called "bug" (e.g., Fraser J in *Bates v Post Office Ltd (No 6: Horizon Issues)* [2019] EWHC 3408 (QB)) and "defect" (e.g., in the Humphrey documents from the Software Engineering Institute of Carnegie Mellon University), as well as "error" (e.g., the Law Commission review). We standardise here on "fault".

[42] [2019] EWHC 3408 (QB).

[43] T20090070, In the Crown Court at Guilford, Trial dates: 11, 12, 13, 14, 15, 18, 19, 20, 21 October and 11 November 2010, His Honour Judge N. A. Stewart and a jury, 12 *Digital Evidence and Electronic Signature Law Review* (2015) Introduction, 44 – 55; Documents Supplement.

[44] [2007] EWHC 5 (QB).

aircraft control system. Such a system consists (crudely) of a computer interposed between the cockpit controls operated by the pilots and the aerodynamic control surfaces of the aircraft: the pilots manipulate their controls; the manipulations are sensed by electronic devices called sensors and communicated to the flight-control computer; the flight-control computer processes these sensor values as input, calculates what has to happen with the aerodynamic control surfaces to achieve the intended command, and then issues detailed commands to the control-surface actuators to move the control surfaces to get that to happen. A failure occurring in that control-logic chain is an event that could, in principle, occur at any time and may endanger the safety of the aircraft. An example is where the aircraft is near the ground, and the pilots issue a "climb" command by pulling back on the control stick/column, but the flight-control computer issues "nose down"/"descend" commands to the elevator actuators.

In each of these processes, whether discrete-demand or continuous operation, the failures occur in effect *randomly* from the point of view of the system logic: they arise through inputs to the processes combined with features of the *system environment* (the relevant parts of the world in which the system sits). The inputs to the processes are generally not predictable to the system (otherwise the values would not need to be input), and the features of the system environment are also just happenstance as far as the system is concerned.

In statistics, phenomena that occur randomly in time are said to form a *stochastic process.* The reason failures are considered to be random is that, as indicated above, the environment is not under the control of the system (it can be too hot, too cold, too damp or wet, too full of ionising radiation, and so on), and one cannot generally say in advance when demands or commands will occur and what they will be. Hence, crucially, it cannot be predicted with certainty from the logic of the system when failures will occur. This means that we are in the realm of probability if we wish to answer questions such as "how reliable is this system?"

Quite simple probability models can be used to address such questions. In the case of a demand-based system, the random sequence of failures is construed to follow a Bernoulli Process. A measure of reliability here is *probability of failure on demand* (pfd). In the case of continuously operating systems, an appropriate model is a Poisson Process. Here a measure of reliability is *failure rate.*

We illustrate how these models look by providing the mathematical definition of a Bernoulli process. First, we need a few more definitions. A *random variable* is a quantity that acquires a *value* during the operation of a stochastic process. We have noted above what values are appropriate for describing the failure process of a discrete-demand system, namely *success* and *failure*. So a specific demand is represented by a random variable, and the value of that variable reflects how the demand turns out. This is called a *Bernoulli trial*. Two trials are said to be (stochastically) *independent* if the probability of a particular value of one trial is completely unaffected by an observed value of the other.

Mathematically, a *Bernoulli process* is a sequence of Bernoulli trials, that is, a finite or infinite sequence of *independent random variables* X_1, X_2, X_3, ..., such that

- for each i the value of X_i is either 1 (standing for *failure*) or 0 (for *success*);
- for all values of i, the probability p that $X_i = 1$ is the same.

An individual random variable X_i here is a Bernoulli trial. Two Bernoulli trials are called *identically distributed* if they have the same p. A Bernoulli process is thus a sequence of independent identically-distributed Bernoulli trials.

Independence here is crucial. It can be shown that, if the probability p is known, past outcomes of previous trials provide no information about the outcomes of any trials in the future. (If p is unknown, however, the past does inform indirectly about the future, in that an observer will be trying to judge the likely value of p through statistical inference from the trials they have observed in the past) With the interpretation of the value of X_i as success or failure, the parameter p is a measure of the reliability of the underlying system which the Bernoulli process is characterising.

For continuously operating systems, failures occur in a Poisson process, which can be seen as the continuous-time equivalent of the discrete-time Bernoulli process. The parameter of this process – the equivalent of p in the Bernoulli process – is the *rate of occurrence of failures*, often denoted λ, which is a measure of a continuously-operating system's reliability. Poisson processes share with Bernoulli processes the characteristic that the history of past failures and their

timing does not affect the probabilities associated with future events, so long as λ is known.

This business of independence of past and future, in both the Bernoulli and Poisson processes, is a mathematical property. Whether it applies to a real system, at least to a close approximation, would need to be checked.

Further details and mathematical properties of these processes can be found in textbooks on stochastic processes, and in many locations on the Web.[45]

It is worth noting – but is beyond the scope of this article to demonstrate – that the Poisson process can, under appropriate circumstances, be used as an approximation to the Bernoulli process. In the case of the Horizon system considered in *Bates v Post Office Ltd (No 6: Horizon Issues)*[46], *R v Seema Misra*[47] and *Post Office Ltd v Castleton*,[48] one could treat the failure events as if they occurred in calendar time as a Poisson process, thus approximating to the Bernoulli process in which they occur in terms of counts of sub-postmaster demands. However, certain phenomena such as "phantom transactions", considered at [287] – [295] of the Technical Appendix to Judgement (No. 6) in *Bates v Post Office Ltd (No 6: Horizon Issues)*,[49] do not occur as a result of real demands (hence the word "phantom" used by Fraser J), but are seemingly-random events occurring in a continuously-running system. These are better modelled using Poisson processes.

A major technical concern is the relationship between *faults* and *failures*. How do faults in software affect its reliability? Common sense might suggest to us that, all things being equal, fewer faults generally result in higher system reliability – in terms of the Bernoulli process, there is a *smaller* probability of failure on demand. We might also think that some faults are "larger" than others, and therefore have a greater deleterious effect upon reliability than "small" faults. So, for a demand-based system, we can consider the *size* of a fault to be the probability that the fault will cause a system failure on a randomly-selected demand. For a continuously operating system the *size* of a fault is the rate at which it would cause a failure – in a Poisson process – during operation of the system.

Some of the issues in the relationship between faults and failures that have been addressed in the software-engineering literature are relevant to our critique of the LC Presumption. We described some of this work below, and its relevance to legal evidence.

A seminal early paper by Adams[50] used a world-wide database of software faults in many thousands of large IBM computer systems to investigate the variation in their sizes. The study involved thousands of computers around the world, with tens of thousands of years of combined operational exposure. For each fault, the duration of its exposure, and the number of times it manifested itself during that time, allowed Adams to estimate its size (the rate of the Poisson process associated with that fault).

Adams' results were surprising to the software-engineering community of the time. It turned out that fault sizes varied enormously: the largest occurred many orders of magnitude more frequently than the smallest. Furthermore, the smallest faults were extremely small.

Adams looked at two classes of software. One was a single software product in three different "releases" (versions). He divided the faults into 8 classes, ranging from those which manifested in failure on average every 30 months of operation, up to faults which manifested in failure on average once in every 95,000 months of operation – a little over 7,900 years. He was able to obtain data on faults which only manifest in nearly 8,000 years of operation because the software was running in many, many places. The significant observation is that the faults he identified in this rare-manifestation class were between a fifth and a quarter of all faults.

In the second class, Adams looked at nine different IBM software products. Here, the rarest manifestation class was once in 60,000 months, that is, once in 5,000 years. The 9 products were uniform in so far that, for each product, approximately one third of all the bugs which manifested were in this once-in-5,000-year class. That is, for one third of the faults in the software

[45] For example, Kyle Siegrist, Random, Chapter 10: Bernoulli Trials, available at https://www.randomservices.org/random/bernoulli/index.html , and Chapter 13: The Poisson Process, available at https://www.randomservices.org/random/poisson/index.html

[46] [2019] EWHC 3408 (QB).

[47] T20090070, In the Crown Court at Guilford, Trial dates: 11, 12, 13, 14, 15, 18, 19, 20, 21 October and 11 November 2010, His Honour Judge N. A. Stewart and a jury, 12 *Digital Evidence and Electronic Signature Law Review* (2015) Introduction, 44 – 55; Documents Supplement.

[48] [2007] EWHC 5 (QB).

[49] [2019] EWHC 3408 (QB).

[50] Edward N. Adams, 'Optimizing preventive service of software products', IBM J. of Research and Development 28(1): 2-14, 1984.

manifested in his study across these nine different products, each led to failures on average every 5,000 years or longer.

The results from the general fault-failure model used by Adams were validated in other empirical studies. NASA had been funding extensive theoretical and experimental studies in software engineering to support their quest for high reliability in deep-space exploration. Under this programme, some ingenious experiments were conducted on some scientific and engineering programs, albeit smaller than those studied by Adams, to measure the sizes of their faults.[51] These experiments, and later ones,[52] produced very similar results to those of Adams concerning extremely large variation in fault sizes.

These data, of course, are several decades old, and computer technology of all kinds has progressed in that time. For example, Adams' "most-frequent" category of failure was once-in-30-months; today one could be reasonably confident to detect all such bugs in pre-release testing by running the software on 1,000 inexpensive desktop computers for a day. Such a scenario seems entirely feasible for a software company. Indeed, it may well be that all categories up to Adams's "least frequent" can be detected today by suitable "test farms" and therefore corrected before market release.

But that still leaves the problem of the remaining faults, ones that manifest very infrequently. It is not possible to be certain that there are not very many of these, particularly in large and complex software systems. The exact numbers may have changed somewhat since the studies were carried out, but we see no reason to think that the qualitative phenomenology of software faults and failures as identified by Adams and others has changed significantly, and our collective experience supports this view.

The model Adams used, that individual faults on continuously-operating software manifest in failures as a Poisson Process, is a general principle which is as applicable today as it was when Adams was writing. Because of the significantly increased power and speed of computers in the intervening time, most of the fault classes which Adams used have become, in principle, detectable with standard pre-release testing. Adams stopped with his "most infrequently manifesting" class at a 5,000-year, or 7,900-year mark. This would be no longer appropriate today. The values that Adams chose were arbitrary cut-offs which he used to summarise his data. It is now reasonable to assume that the "left tail" of the distribution of fault sizes extends even further to the left than Adams' cut-off points. If Adams were to be writing today, the "window" of fault classes would simply have shifted to more-infrequent faults.

Discussion of the relevance for the legal profession

We consider the general conclusions that can be drawn from these empirical and experimental results. In particular, we contemplate what they tell us about the appropriateness of the LC Presumption. We suggest three tropes that lawyers, judges and academics writing on evidence may wish to keep in mind when constructing and judging arguments concerning the dependability of software.

Consider the statement of the Law Commission at 13.23: "We are satisfied that the assumption of proper functioning would apply to computers…"

This seems to us to come close to asserting that it should be assumed that a software fault is not the cause when some untoward event has occurred, unless there is overt evidence of such a fault. Clearly such an assertion could not be supported when the software in question has been shown to be unreliable. But what about the case when the software is demonstrably reliable, as shown, for example, by empirical evidence such as extensive failure-free working? Here, the lesson from the Adams data (discussed above), and others, is that even software like this may have – indeed is likely to have – latent faults (probably very small ones) that will eventually show themselves as failures during operation. (As well as latent faults that will not so show themselves.) In the absence of evidence to the contrary, therefore, it would be unreasonable to assume that in a particular instance software is innocent of causing failure. Thus:

(1) A presumption that any particular computer system failure is not caused by software is not

[51] Phyllis M. Nagel and James A. Skrivan, *Software reliability: repetitive run experimentation and modelling*, Boeing Computer Services Company, NASA-CR-165836, February 1982, available at https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19820013026.pdf.

[52] Janet R. Dunham and John L. Pierce, *An experiment in software reliability*, NASA Langley, NASA-CR-172553, May 1986, available at https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19860020075.pdf.

justified, even for software that has previously been shown to be very reliable.

The text of 13.23 goes on to provide a "let out" for the defence from the assumption of proper functioning of software. Essentially this states that the defence should not necessarily fail if it had not been in a position to find evidence of malfunction on the part of the software (and thus had, presumably, not found such evidence). This appears to allow the introduction of indirect evidence – i.e., short of direct evidence of malfunction in the particular instance under trial.

We provide a couple of examples of the kind of indirect evidence that might be appropriate.

An obvious example would be evidence that the software had previously "failed similarly", so that one might infer this may have happened in this instance, even though there was no direct evidence for this.

For a second example, consider a case in which there is agreement that some untoward event is caused either by software, or by a human operator. If there have been failures in the past where it has been agreed which of these was the cause, then the failure process is a super-position of two independent Poisson processes: one of computer failures and one of human failures. (The overall human + software failure process can also be shown to be a Poisson process.) Rigorous statistical arguments can then be used to claim how likely it is that, in this particular instance, the responsibility lay with the computer rather than the human. Again, a situation like this would seem to undermine the "proper functioning" presumption of 12.23. Thus:

2) Evidence of previous computer failure undermines a presumption of current proper functioning.

How should we consider a case where there has been no evidence of occurrence of a software failure that masquerades as a human failure (for example, the "phantom transactions" considered by Fraser J)? Can or should it be concluded that such a failure cannot occur?

It is well known amongst software engineers that software can fail in ways that have not been seen before. Most obviously, a new fault manifests itself for the first time, and the consequences – the nature of the failure – are different from those that have been seen in previous failures. The Adams data show that there can be many such latent faults.

More formally, consider a possible class of faults that has not been seen to manifest itself yet; for example, human-masquerading faults as considered in the example above. If we assume that all faults manifest themselves as Poisson processes, then it can be shown that any subclass of such faults will also occur in a Poisson process. So how confident should we be that we shall not see a manifestation of a human-masquerading fault in the future, if we have not seen one in the past? This problem was addressed in some detail in a paper by one of the authors and his colleague.[53] The informal answer is that we cannot be very confident: for example, if we have seen the system in operation for *x* hours without failure there is only about a 50:50 chance that it will survive a further *x* hours before failing. This means that if a particular class of faults has not shown itself even in massive testing, our confidence in it not manifesting in operation can only be rather modest.

It follows that:

(3) The fact that a class of failures has not happened before is not a reason for assuming it cannot occur.

[53] Bev Littlewood and Lorenzo Strigini, 'Validation of ultra-high dependability for software-based systems', Comm. ACM 36(11): 69-80, 1993. https://openaccess.city.ac.uk/id/eprint/1251/.