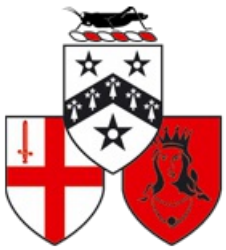

GRESHAM COLLEGE

Founded 1597

Should We Trust Computers? Transcript

Date: Tuesday, 20 October 2015 - 6:00PM

Location: Museum of London

20 October 2015

Should We Trust Computers?

Professor Martyn Thomas

Good evening. It is an honour and a privilege to have been appointed as the first *Livery Company Professor of Information Technology at Gresham College* and I would like to thank the Worshipful Company of Information Technologists and the Academic Board of Gresham College for making it possible for me to be here to speak to you this evening.

I have spent my working life as a software engineer, mainly in the private sector but always with strong links into universities and Government. I have set up and run a company, *Praxis,* that developed software for computer companies, banks, education, industry and commerce. I have also been a partner in Deloitte Consulting and an expert witness in major court cases that involved computer systems, in Europe and Australia. My work has taken me behind the scenes in a variety of different organisations, ranging from retailers and banks to air traffic control, the police and the Ministry of Defence. In short, I have seen a lot of software projects, developed by a lot of different companies, and I have some serious concerns that we are already taking unreasonable risks and that these risks are growing.

This evening, I shall start to explain why I am concerned. I hope to use the next three years at Gresham College to discuss the issues with you and to explore the best way forward. Everyone in the hall this evening is a computer user, through your phone, your contactless payment card, your car, your television and much more. So whether you are directly involved in designing computer systems or not, you have a stake in how information technology is used today and how it will develop over the next decade and beyond.

The power of computing and the inventiveness of engineers and entrepreneurs have brought about radical changes to the way we all live. Even for those of us in the computing industry, it has been difficult to foresee the consequences of each new technology. It was inevitable that regulation and legislation would fail to keep up with the challenges of cybersecurity, big data analytics, robotics and autonomous vehicles. The pace of change is not slowing – indeed it seems to be accelerating – so I think we should see whether we, the citizens of cyberspace, can influence the design of our future world.

Gresham College is ideally placed to convene and host discussions about how we want to use information technology in future. We have four centuries of tradition in holding public lectures and discussions, and very close links with the City of London, which is now so dependent on computing.

We can hold our discussions both here and also in cyberspace. We have set up a website, *http://www.cyberliving.uk* to host discussions and if you use the hashtag *#cyberliving* on Twitter I will try to incorporate those tweets into our discussions. This means that those of you watching the recording of this talk on the internet can also join in the questions and discussions. Please do join it, and please let's keep any arguments polite and constructive. This is new ground for Gresham College, so let's see how well we can make it work.

I hope you will help me tonight and throughout this series of lectures, as we explore and discuss some of the difficult issues that already exist and some of the difficult decisions that will have to be taken in the next few years and beyond. In the near term, we shall see new data protection legislation across Europe, and new UK laws about surveillance, intelligence and hacking, after Edward Snowden's revelations about the activities of the US National Security Agency and the UK's GCHQ. We shall see the installation of 53 million smart gas and electricity meters across the UK, with new tariffs and, later perhaps, the ability for electricity suppliers to control our fridges and freezers, heating, air conditioning and the chargers for our electric vehicles, so that they can be turned off if demand gets too high. We may soon see driverless cars and lorries on our streets and motorways, perhaps with far-reaching consequences. Further ahead, there will be decisions to be made about whether we need controls over artificial intelligence and (perhaps surprisingly soon) artificial life.

But first I want to examine the state of computing today, and whether we are already putting too much trust in computer systems and in the people who develop them.

Information technology pervades our lives. If I could clap my hands and switch off all computers, the lights would go out, trains would stop, taps would run dry, food would not be delivered to shops, cars and aircraft would crash, telephones would be dead, hospitals would cease to function, no money would be available from banks, most shops would be unable to accept payment (and those that could would soon run out of goods to sell). There would be riots in cities within hours and the emergency services would be helpless. In short, the disruption would be enormous and very many people would die.

This is not an exaggeration. We are completely dependent on the modern electronic computer and software – technologies that are only two months older than I am myself.

The first modern computer was invented and built in Manchester University by F C (Freddie) Williams, Tom

Kilburn and Geoff Toothill and ran its first successful program on June 21 1948.

Computing has transformed the world in the last 67 years. We have millions of different devices and applications.

Computer hardware – the physical components that store the information, carry out the processing and communicate with us and with other computers – all of that has become extraordinarily small, fast, efficient and cheap.

There have been extraordinary changes during my own career.

I started using computers myself in 1969 as a biochemistry student at University College London.

UCL had one large computer – an IBM 360/65 that was said to have cost over £2m (which would be more than £30m in today's money). It had 128K of main memory and executed instructions at the (then) amazing rate of 2 million per second. It provided all the computing services for many hundreds of students and staff. Today you have hugely more power and memory in your mobile phone.

Computer processors have doubled in power or halved in price every two years since the mid-1960s, and disks and memory have followed much the same path. It is these hardware advances that have made computers affordable in the enormous range of different applications, and led to computers coming out of the large, air-conditioned computer rooms of the 1960s and appearing in departments, then on desks, and now inside telephones, cars, toys and light-bulbs.

In the whole of human history, no other invention has changed so many lives, so quickly.

Yet we are only at the beginning of the revolutionary changes that computers will bring to society. We are on the verge of the "internet of things", where almost everything could contain intelligence and be network connected.

Autonomous air vehicles with high resolution cameras and satellite navigation have already moved from military applications into toyshops – and at least one of them has been modified to fire a handgun.

It is the dramatic improvements in computer hardware that have driven the economics of this new industrial revolution.

But the technology that has created most of the huge and complex diversity of uses is *software*: computer programs. Software provides the variety and flexibility; software contains the complexity of all the different things we want to do.

Software is almost everywhere! And there will be lots *more* software written, because we are still only at the beginning of this **information age**.

Because modern life depends so heavily on software, it seems prudent to consider how dependable that software is today and how we should feel about proposals for new computer systems that could cause serious harm if they go wrong. How confident should we be that the software on which we already depend is adequately reliable, safe and secure? And how confident should we be that other new and critical systems can be built at reasonable cost and with reasonable success?

After four decades of work in the software industry, I am increasingly nervous and so are many of my colleagues because, despite the tremendous achievements of the computing industry, all this progress is dependent on a software industry that is still very immature – still a long way from becoming a mature engineering profession, even though it is 45 years since the phrase *"software engineering"* first came into common use.

Most computer programmers do not have qualifications in computer science or familiarity with the rigorous management disciplines that are fundamental to every engineering profession. The software industry is brilliantly creative and agile and brings new ideas to market with extraordinary speed. But there is a big difference between a product that works well most of the time and one that is truly dependable. When the product needs to be shown to be particularly safe or secure, speed and agility should take second place to those critical properties, but building safe and secure computer systems is difficult and can be expensive, and too few programmers know how to *show* that a system is adequately safe or secure. As we shall see later, testing alone can never be enough.

Designing and building trustworthy systems is an engineering task that requires methods and tools with strong scientific foundations, supported by processes that manage complexity, that prevent careless and unsafe ways of working that control and manage change, and that deliver sufficient evidence that the resulting product is dependable. Without strong engineering methods, it is highly probable that the development will suffer major overruns in costs and time, or that it will fail to do the job it was supposed to deliver, or that the resulting software will contain very many defects that leave its users and others in society exposed to cybercrime.

I would like to explain just how faulty much software is today, and remind you of some of the things that have gone wrong.

Almost all software contains a lot of faults

Software quality is generally poor. Academic studies have shown defect rates between 10 and 30 defects in every 1000 lines of program source.

In the USA, Steve McConnell has studied software defects for many years. His book *Code Complete* **[1]** says that the industry average is about 15 - 50 faults per 1000 lines of delivered code. Of course, some companies do better than this; one example is Microsoft, which stopped work to retrain its programmers over a decade ago and invested in many, world class development tools. But if you are delivering software that contains 50 million lines (which is said to be the size of Microsoft Windows) then even *one* fault in each thousand lines would mean that you are selling software that contains 50,000 faults.

The American author, trainer and consultant Capers Jones has published substantial data on delivered product fault levels [2]. In his sample, the most quality-conscious companies delivered systems with an average of about 1 fault per thousand lines, whereas other companies in his sample were delivering over 7 faults per thousand lines. (Different authors use different definitions of what constitutes a fault or how to count lines of software so the figures from different authors are not wholly comparable. The key message from everyone who has studied defect rates is that software contains a large number of faults).

One detailed analysis of defence software, a paper called *Software Static Code Analysis Lessons Learned* **[3]** was published in *Crosstalk*, the journal of defence software engineering, by Andy German, a software engineer in QinetiQ who worked on military aircraft systems at the UK Ministry of Defence at Boscombe Down.

He analysed a range of software systems that had been developed for an American military aircraft. The systems included:

- Automatic flight control.
- Engine control.
- Fuel and centre-of-gravity management.
- Warning systems.
- Anti-icing systems.
- Flight management.
- Weapons management.
- Air data units.
- Radio altimeters.
- Anti-skid brakes.

These systems varied in size from 3,000 lines of code to 300,000 lines of code and were written in several different languages. He reported that he found the following level of "anomalies".

Table 1: *Software Language Anomaly Rates*

Software Language	Range	Software Lines of Code Per Anomaly	Anomalies Per Thousand Lines of Code
C	Worst	2	500
	Average	6 - 38	167 - 26
	Best (Auto Code Generated)	80	12.5
Pascal	Worst	6	167
	Average/Best	20	50
PLM	Average	50	20
Ada	Worst	20	50
	Average	40	25
	Best (Auto Code Generated)	210	4.8
Lucol	Average	80	12.5
SPARK	Average	250	4

The developers were following the then current standard for civilian aircraft software DO-178A, because the aircraft had to be certified to fly in civil airspace.

Again, I do not want to focus on the definition of an "anomaly" or how lines of code are defined (when a single line in a program listing could contain several program statements or none); I am simply illustrating that every detailed study finds that there are a lot of defects in most software.

Another study, this time by Watts Humphrey, of the Software Engineering Institute at Carnegie-Mellon University, analysed the fault density of more than 8000 programs written by 810 industrial software developers [4]. This was the result:

Table 25-1: Defect Injection Rages for 810 Experienced Software Developers

Group	Average Defects per KLOC
All	120.8
Upper Quartile	61.9
Upper 10%	28.9
Upper 1%	11.2

A "KLOC" is 1000 **L**ines **O**f **C**ode. These figures show that even the best 1% of these programmers were introducing a fault in every 100 lines of program that they wrote. The average programmer was introducing a fault in every 9 lines.

Different researchers and authors may describe faults as "flaws", "errors", "defects", "anomalies" or "bugs" but they will almost always mean *functional* faults, which cause the software to crash or to give the wrong results. A functional error may be a simple typing mistake, such as omitting a minus sign in a mathematical formula, or testing that value A is *greater than* value B [A>B], when they should have written a test that value A is *less than* value B [A<B] .

Or it may be a design error that means that the program sometimes attempts to divide by zero, or runs out of the available memory.

Watts Humphrey distinguishes between *functional* errors and *security* errors. In the same CMU Technical Report[5] he writes:

Today, most programmers are unaware of many types of security defects. While some security defects are generally recognized as functional defects, others are more subtle. A security defect is any design fault that permits hackers, criminals, or terrorists to obtain unauthorized access or use of a software system. Since many of these defects do not cause functional problems, systems that are riddled with security flaws may work just fine and even pass all of their functional tests. And, since many programmers are worried only about getting their programs to function, they are almost totally unaware of their products' potential security vulnerabilities.

In one program that had been supporting a Web site for over two years, a security audit found one functional defect and 16 security defects. So, today's one MLOC systems with 1,000 functional defects could easily have many thousands of security defects, and almost any of these could provide a portal for a hacker, thief, or terrorist to steal money, disrupt records, or to otherwise compromise the accuracy of the organization's

financial system.

In this particular article, Watts Humphrey was focusing particularly on the risks to financial systems and on the responsibilities of company directors and senior managers, but similar security faults and exist in computer systems that control vehicles, factories and physical infrastructure, where the consequences could include fatalities and environmental damage.

We shall look at some typical cybersecurity faults – and at how hackers exploit them – in a future talk, next May; the key message from today is that most faults that are exploited in cybersecurity attacks result from inadequate (I would even say incompetent) software engineering that we can no longer afford to tolerate.

Software is big and getting bigger

The following table is taken from the spreadsheet underlying the visualisation at http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Product	**Lines**
Simple smartphone application (imangi)	10K
Photo editing smartphone application (Camera Genius)	40K
Heart pacemaker	80K
Photoshop (version 1)	120K
Space Shuttle	400K
Age of Empires online computer game	1.2M
F22 Raptor fighter aircraft	1.7M
Microsoft Windows 3.1 (1993)	4.3M
Photoshop CS6 (2012)	4.5M
Mars Curiosity Rover	5.0M
Linux kernel 2.6.0 (2003)	5.2M
Firefox web browser	9.7M
Boeing 787 flight software	14M
Microsoft Office 2001	25M
Microsoft Office for Mac (2006)	30M
Microsoft Windows XP (2001)	40M
Microsoft Office 2013	45M
Facebook (without back end code)	62M
Mac OSX 10.4	86M
Software in a typical new car	100M
Healthcare.gov website (Obamacare) 'according to the NY Times'	500M

Although the most recent Microsoft Windows development probably has a significantly lower fault density than earlier, Windows will still contain a lot of software that was written 10 or 20 years ago.

The Boeing flight software will have been developed using better engineering methods than is typical for commercial or industrial software—because a very disciplined approach is required by the relevant standards and the Federal Aviation Administration—but the flight software that Andy German analysed for his *Crosstalk* article and whose fault density table I have already shown was also developed to those standards. When software contains millions of lines, we need fault densities to be far, far lower than 10 faults per thousand lines or even 1 fault per thousand lines. Otherwise we can be sure that there will be very many faults and very many security vulnerabilities in the product.

Computer scientists and software engineers know that it is possible to design and develop software so that it is almost completely free of faults. You can even prove mathematically that the system will never crash as a result of a software fault (which provides strong assurance for the absence of many whole classes of security fault). A few companies work this way and, as I will show in a later talk, it doesn't have to make the software more expensive; better can also be cheaper.

Many software developers scorn the use of mathematically rigorous development methods, claiming that they can only be used on trivially small problems but they are wrong; it is perfectly practical to use rigorous software engineering methods on big projects. Examples where these *formal methods* have been used include the signalling for the Paris Metro, developed by Siemens, and the air traffic management system (iFACTS) that Altran recently developed for the UK's National Air Traffic Services, NATS.

The iFACTS system is about 250,000 lines of code, written in a programming language called SPARK [6]. To show that the software cannot generate any exceptional conditions that could lead to a run-time failure, Altran needed to carry out more than 152,000 individual proofs of *verification conditions*. The SPARK tools were able to prove

98.76% of these verification conditions automatically, and the remainder with help from the software engineers.

To prove the whole system of 250,000 lines takes 3 hours on a 16-core desktop computer with 32Gb RAM (in other words, a small computer costing only £2,000), and the SPARK prover can save and reuse successful proofs, with the result that re-proving the system after a change can be done in 15 minutes. [These figures come from a paper written by the software engineer Rod Chapman who developed and supported the SPARK software tools. The paper is available online [7] and the SPARK tools can be downloaded and used free, as I will explain in a future lecture].

Unfortunately, most software developers believe that it is reasonable to rely on testing rather than analysis to show that their software is fit for purpose. Indeed, when a computer system crashes, you often hear people saying that it wasn't tested enough. It is easy to show that this is nonsense.

For example, consider a program that contains two integer variables, *a* and *b*. [A *variable* is a named place in the computer's memory that contains some data—in this case a whole number or *integer*]. Suppose that the program calculates the result of the expression *1/(a-b)*. You might run and pass a million tests with a million different numbers placed in *a* and *b* but then, the first time the number in *a* is the same as the number in *b, a-b* will be zero and the attempt to divide by zero could cause the program to fail. The million successful tests have given you no evidence that the system could not fail.

Of course, a large program will contain many hundreds of variables and thousands of calculations any of which could fail. If you run a million different tests without any failures, you will have tested a negligibly small fraction of the possible combinations, and learnt nothing of value about the probability that your program will fail in the future [8]. So when you hear someone say "it wasn't tested enough", ask them "how much testing *would* have been enough?". In contrast, the SPARK Analyser can tell you with certainty that your program will never fail in this way – or it will show you where it could.

That is a trivial example, but enough to show the limitations of testing. In reality, programs can go wrong in many different ways and no possible amount of time and money invested in running tests can deliver the sort of assurance that we need for really important software systems. Really high confidence can only come from analysing the software logically, to prove that it has the properties that we need [9], and we need to choose our programming tools and methods very carefully to make that analysis possible. I shall return to this theme in later talks. Of course, software faults can be much more complex than a simple zero-divide. But even a zero-divide can be very damaging. In July 2015, a zero-divide error was reported in the Android software that is used by most smartphones other than the iPhone. Sending someone a multimedia text message containing a special MKV video file will cause the phone to crash. This was estimated to affect 90% of Android phones. (A further fault in the Android video software means that a criminal can install their own software on your phone just by sending you a message. All they need is your phone number – and you do not even need to click on the message).

Such problems are not new. We have had computer viruses and worms for decades and some of them have caused huge damage and costs.

The 1999 Melissa virus [10] is estimated to have caused damage exceeding $1B. It propagated as a Microsoft Word document containing a malicious macro which sent a copy of the same dangerous document to all your contacts. Why did Microsoft think it was a good idea to design Word so that a file that appeared to be a text document would be capable of reading your contacts list and sending emails? Or, at the very least, why didn't they write the software so that it had to tell the user what it was about to do, and ask for permission?

The 2003 Slammer worm [11] also caused over $1B damage. According to the referenced report from the International Computer Science Institute at the University of California, Berkeley: *"it spread so fast that it infected more than 90 percent of vulnerable hosts within 10 minutes, causing significant disruption to financial, transportation, and government institutions and precluding any human-based response"*. Slammer exploited a fault (in a commonly installed database product) that allowed an attacker to overwrite the program with their own software. This class of faults – *buffer overflows* – should be impossible in any programming language that is meant to be used for important software development. Unfortunately, most programmers choose to use programming languages that make it easy to make errors such as this one.

Software faults have also led to fatalities. One that has been analysed thoroughly by Professor Nancy Leveson is the Therac-25 radiotherapy system that massively overdosed six people [12].

So there have always been serious consequences from inadequate software development, but recent developments have increased the threat dramatically. The early safety problems were the chance outcomes of accidental errors (and some incompetence) and the viruses and worms were motivated mainly by vandalism or curiosity. More recently, organised criminal groups have recognised that it is safer and potentially much more profitable to commit crimes online and from overseas than to commit crimes in person. So the data that enables identity theft and fraud or blackmail have become a major target, as we have seen recently with the attack on the US Office of Personnel Management (that stole the intimate and very sensitive records from security interviews for millions of US personnel whose role required security clearance) and the attack on the infidelity website *Ashley Madison.*

An increased focus of security concern and academic research has been *cyber-physical systems*; these are computer systems where faults or cyber-attacks can cause serious problems to physical systems such as oil refineries or vehicles. The threats that exist here from faulty engineering are legion and I do not want to go into any detail today, but the recent demonstration[13] that a Chrysler car could be disabled or controlled remotely over a mobile communications network has led to the recall of 1.4 million cars[14] and emphasises the urgent need for far better engineering certification before driverless cars are licensed for use on public roads.

The final consequence of the immaturity of the software industry that I want to highlight today is that software projects often overrun in time and cost. Software developers are either very poor at estimating or managing their projects, or they lie to their customers (or quite often, in my experience, both). Billions of pounds have been wasted on failed software projects in the UK, in industry (and also in Government as a search of the reports from the National Audit Office and the Public Accounts Committee will confirm). I shall look at some of the reasons for this and some of the solutions in my lecture next February.

As I said in opening, Information Technology has changed the world in less than 70 years. The world in 2015 is startlingly different from the world in 1948 and much of the difference has been brought about through computers and software.

But the pace of change shows no signs of slowing down and there are decisions to be made.

Can we tolerate the way that companies and Governments collect and analyse our personal data, so that they know more about us than we know ourselves?

What is the right balance between protecting the privacy of sensitive medical histories of patients and the benefits that could flow from using the records for research? What should we do about the growing problems of cybersecurity?

What is the right balance between privacy for individuals and the need to detect crime and catch criminals? Are the fears that have been expressed about artificial intelligence outstripping human intelligence justified?

These are really questions about the sort of future we want for ourselves and our children, and I believe the decisions are too important to be left entirely to companies in their competition for profits. I would like to see discussions and debates, informed by a good understanding of the nature, current capabilities and future potential of information technology, so that we can influence industry and Government towards the sort of future we would prefer. I hope that together we can use these lectures and Gresham College to have some of these discussions. My planned lectures for the rest of this academic year have been published in your programme booklet and on the Gresham College website. I have already mapped out possible lectures for the following year but I welcome your views on the topics you most want to hear about and to discuss.

Thank you for coming today. I hope you will come to future talks and I look forward to our discussions.

[1] *2nd Edition.* Redmond, Wa.: Microsoft Press, 2004

[2] Jones, C. Software Assessments, Benchmarks, and Best Practices. Reading, MA: Addison Wesley, 2000

[3]http://static1.1.sqspcdn.com/static/f/702523/92927...

[4] http://resources.sei.cmu.edu/asset_files/SpecialReport/2009_003_001_15035.pdf page 132

[5] ibid page 145

[6] S*PARK: the proven approach to High Integrity Software*, John Barnes 2012. ISBN 9780957290518

[7] http://proteancode.com/wp-content/uploads/2015/05/...

[8] If the tests are truly random then, with strict assumptions about the degree to which the tests represent the way the software will be used in future, you can calculate a failure probability. This will rarely be practical or adequate and the tests must be repeated if the software or its usage are changed.

[9] I am not suggesting that the software should not be tested as well as analysed. But finding a fault that the analysis missed should be regarded as very serious indeed, and lead to a thorough investigation.

[10] https://www.cert.org/historical/advisories/CA-1999...

[11] http://www.icsi.berkeley.edu/pubs/networking/insid...

[12] http://sunnyday.mit.edu/papers/therac.pdf

[13] http://www.wired.com/2015/07/hackers-remotely-kill...

[14] http://gizmodo.com/chrysler-recalls-1-4-million-ca...