cybersecurity centre
Academic Centre of Excellence

CyberSecuritySoton.org [w]

@CybSecSoton [fb & tw]

comp6224 (2016)
week 8: Assurance

Vladimiro Sassone
Cyber Security Centre
University of Southampton

Security assurance and evaluation

design methodologies: the waterfall model

the waterfall model

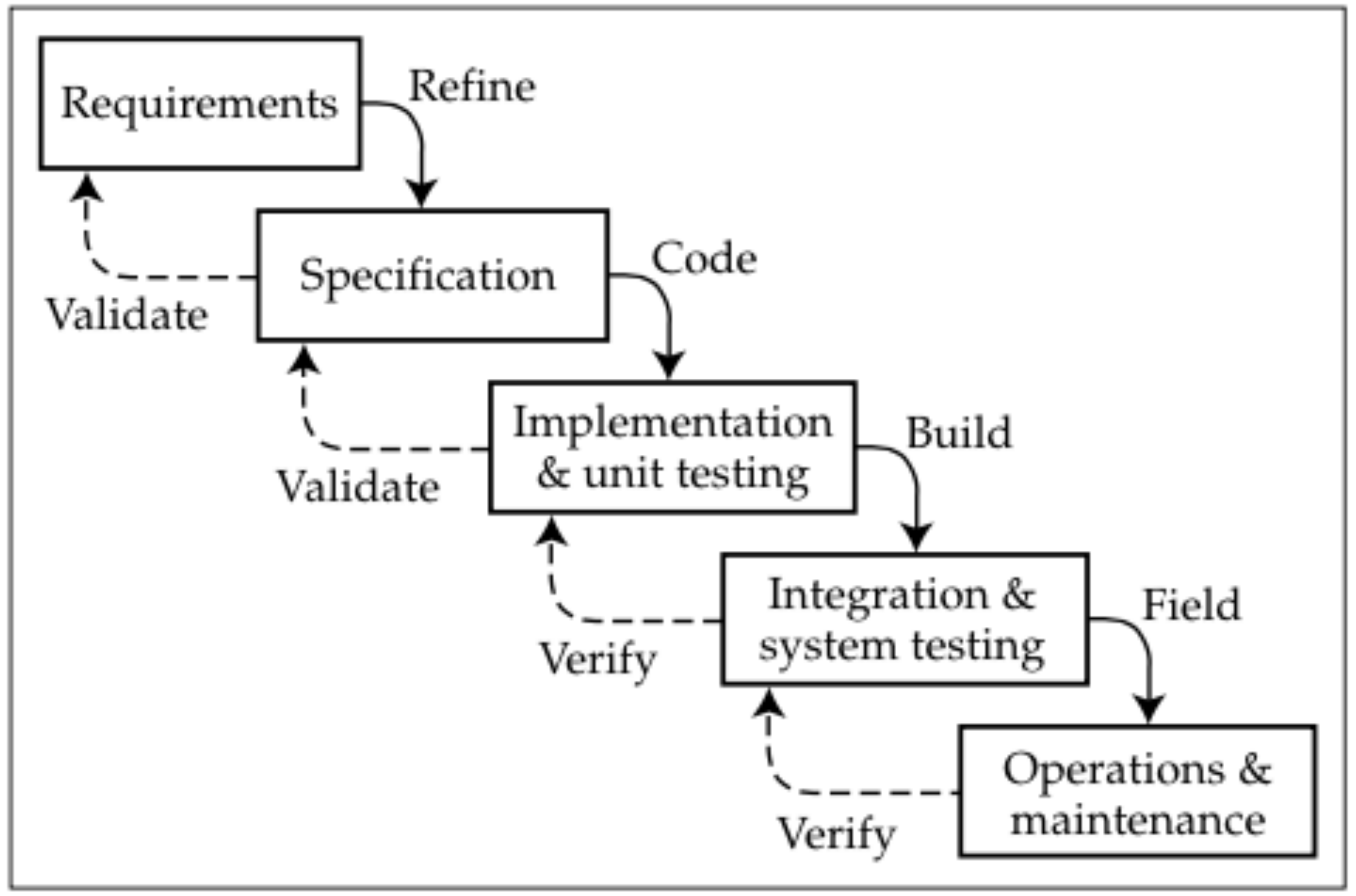

Figure 25.1: The waterfall model

the spiral model

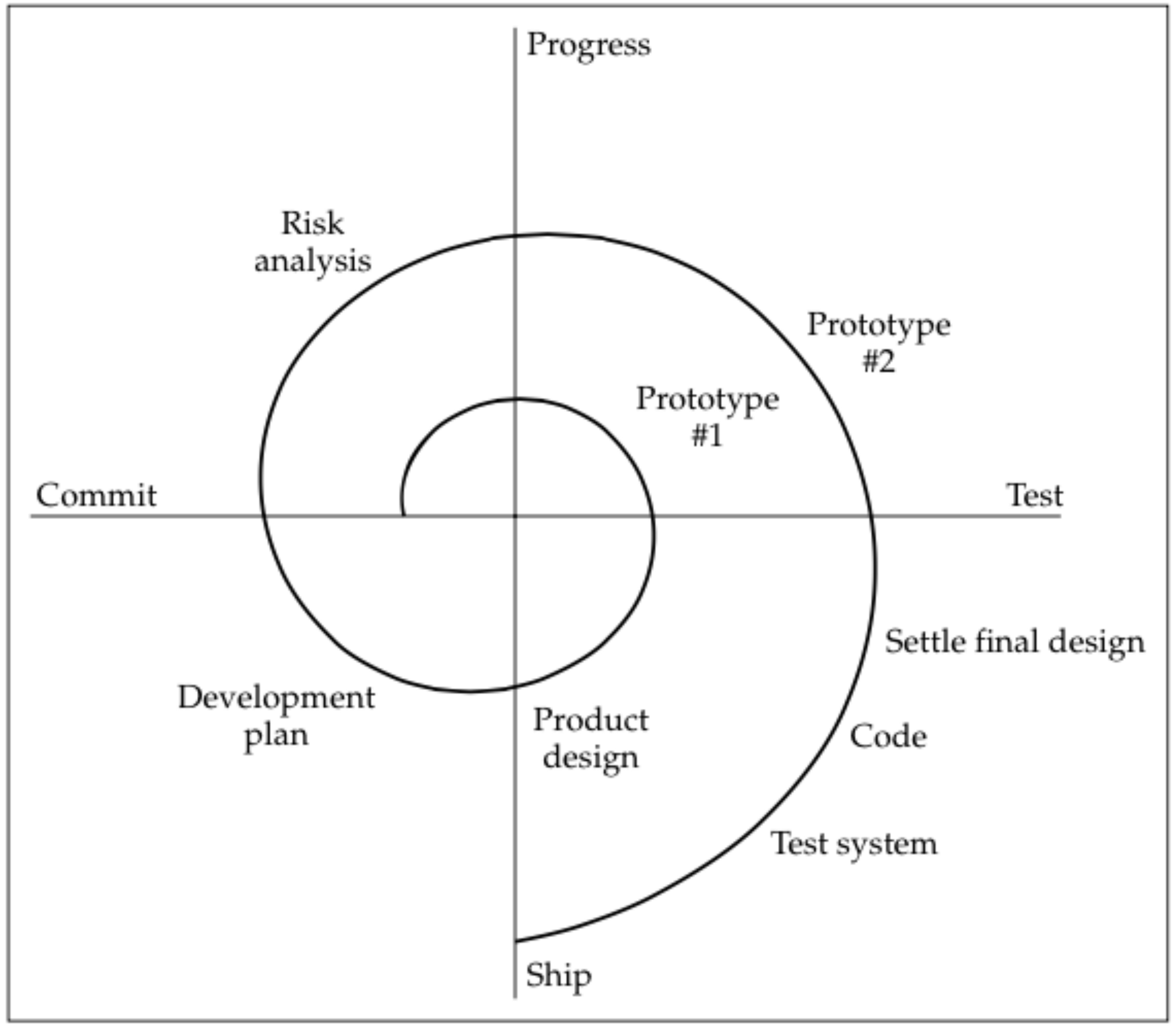

Figure 25.2: The spiral model

security assurance and evaluation

The hardest parts to the last. These are the questions of

assurance: whether the system will work; and

evaluation: how you convince other people of that.

Evaluation is both necessary and hard. Often one principal carries the cost of protection while another carries the risk of failure. This creates a tension, and third-party schemes such as the *Common Criteria* are necessary for transparency.

A working definition of *assurance* could be '*our estimate of the likelihood that a system will fail in a particular way*'.

This estimate can be based on a number of factors, such as

- the process used to develop the system;
- the identity of the person/team who developed it;
- particular technical assessments, e.g., the use of formal methods or the deliberate introduction of bugs to see how many of them are caught by the testing team;
- and experience

It ultimately depends on having a model of how reliability grows (or decays) over time as a system is subjected to testing, use and maintenance

perverse economic incentives

The big missing factor in the traditional approach to evaluation is *usability*. This is crucial as most system-level failures have a significant human component. Designers often see assurance simply as an absence of technical bugs, without stopping to consider human frailty.

Example:

A rational PC user might want:

high usability, medium assurance (it's expensive, and we can live with the odd virus), high strength of mechanisms (they don't cost much more), and simple functionality (usability is more important).

Commercial platform vendors go for:

rich functionality (each feature attracts some users, rapid product versioning prevents a commoditised market), low strength of mechanisms, low implementation assurance, and low usability (programmers matter much more than customers as they enhance network externalities).

perverse economic incentives (2)

Companies racing for dominance in platform markets start out by shipping too little security, as it gets in the way of complementers to whom they must appeal. Once a firm has achieved dominance in a platform market, it will add security, but of the wrong kind, as its incentive is now to lock its customers in.

So vendors provide less security than a rational customer would want; and, in turn, customers want less than would be socially optimal, as many of the costs of attaching an insecure machine to the Internet fall on others.

Government agencies' ideals are also frustrated by economics. They would like to be able to buy commercial off-the-shelf products, replace a small number of components (such as by plugging in crypto cards to replace the standard crypto with a classified version) and just be able to use the result on defense networks: that is, multilevel, or role-based, functionality, and high implementation assurance.

security testing

Security testing usually comes down to reading the product documentation, then reviewing the code, and then performing a number of tests.

First look for any *architectural flaws:*

Does the system use guessable or too-persistent session identifiers? Is there any way you can inject code? Is the security policy coherent, or are there gaps in between the assumptions? Do these lead to gaps in between the mechanisms where you can do wicked things?

Then look for *implementation flaws:*

stack overflows and integer overflows. This will usually involve not just looking at the code, but using specialist tools.

Then work down a list of *less common flaws:*

if the product uses crypto, look for weak keys and poor random-number generators; if it has components with different trust assumptions, try to manipulate the APIs between them, looking for race conditions and other combinations of transactions that have evil effects.

product assurance

Formal Methods

these are good, often required by higher level assurance, but not conclusive.

Monitor the quality of your testing, eg by ***fault injection***:

introduce a number of random errors deliberately into the code. If there are a hundred such errors, and the tester finds seventy of them plus a further seventy that weren't deliberately introduced, once the remaining deliberate errors are removed you might expect that there are thirty bugs left that you don't know about.

For a mature product, you should have a good idea of

- the rate at which bugs will be discovered,
- the rate at which they're reported,
- the speed with which they're fixed,
- the proportion of your users who patch their systems quickly,

and thus the likely vulnerability of your systems to attackers who either develop zero-day exploits , or who reverse your patches looking for the holes you were trying to block.

process assurance

Testing is an assurance approach focused on the product. More emphasis is now put on process measures such as who developed the system. Some programmers produce code with an order of magnitude fewer bugs than others, and some organisations that produce much better quality code than others.

In the 1980s, the waterfall model of system development meant that

- one team wrote the specification,
- another wrote the code,
- yet another did the testing (including some bug fixing),
- while yet another did the maintenance (including the rest of the bug fixing).

They communicated with each other only via the project documentation. The effect was that the coders produced megabytes of buggy code, and poor testers had to clear it up.

Microsoft considers that one of its most crucial lessons learned as it struggled with the problems of writing ever larger programs was to have a firm policy that

‘*if you wrote it, you fix it*’.

Bugs should be fixed as soon as possible; and even though they’re as inevitable as death and taxes, programmers should always strive to write clean code.

the capability maturity model

The *Capability Maturity Model* (CMM) from Carnegie-Mellon University strives for a holistic assessment of a team's capability. It is based on the idea that competence is a function of teams rather than just individual developers.

This research showed that newly-formed teams tended to underestimate the amount of work in a project, and also had a high variance in the amount of time they took; the teams that worked best together were much better able to predict how long they'd take, in terms of the mean development time, but reduced the variance as well.

CMM offers a certification process whereby established teams may get themselves considered to be assets that are not to be lightly squandered.

It has five assurance levels

initial, repeatable, defined, managed and optimising

with a list of things to be added as you go up. Thus, `project planning' must be introduced to move up from 'initial' to 'repeatable', and peer reviews to make the transition from 'repeatable' to 'defined'.

international standards: iso9001

An even more common process assurance approach is the *ISO 9001 standard*.

Its essence is that a company must document its processes for

design, development, testing, documentation, audit and management control.

There is a whole industry of consultants helping companies get ISO 9001 certification.

This can provide at best a framework for incremental process improvement; companies can monitor what goes wrong, trace it back to its source, fix it, and prevent it happening again.

But very often ISO 9001 is just an exercise in ass-covering and box-ticking.

assurance growth

Yet, most customers are not so much interested in the development team as in its product. But most software nowadays is developed by continual evolutionary enhancement rather than in a one-off project.

What can usefully be said about the assurance level of evolving products?

Quality here reaches equilibrium if the rate at which new bugs are introduced by product enhancements equals the rate at which old bugs are found and removed.

There's quite a lot known about reliability growth: where the tester is trying to find a single bug in a system, a reasonable model is the *Poisson distribution*:

probability the bug is *undetected* after t statistically random tests is $p = e^{-Et}$

where E depends on the proportion of possible inputs.

Extensive empirical investigations have shown that in large and complex systems, the likelihood that the tth test fails is proportional to k/t for some constant k.

So the system's reliability grows very much more slowly than thought, achieving a *mean time between failure* (MTBF) of about t/k.

Thus reliability grows linearly with testing time: '*If you want a mean time between failure of a million hours, then you have to test for (at least) a million hours*'

assurance growth (2)

The rule that you need a million hours of testing to get a million hours MTBF is inescapable (up to some constant multiple which depends on the initial quality of the code and the scope of the testing).

This amounts to a proof of a version of ‘*Murphy’s Law*’:

that the number of defects which survive a selection process is maximised.

That is: software testing removes the minimum possible number of bugs, consistent with the tests applied.

If a population of rabbits is preyed on by snakes then they will be selected for alertness rather than speed. The variability in speed will remain, so if foxes arrive in the neighbourhood the rabbit population’s average running speed will rise sharply under selective predation.

More formally, the *fundamental theorem of natural selection* says that a species with a high genic variance can adapt to a changing environment more quickly. Similarly, complex software exhibits the maximum possible number of bugs when you migrate it to a new environment.

Also, each bug’s contribution to the overall failure rate is independent of whether the code containing it is executed frequently or rarely: code that is executed less is also tested less.

evolution and security assurance

Suppose a complex product such as Windows Vista has 1,000,000 bugs each with an MTBF of 1,000,000,000 hours. Suppose that A’s job is to break into the U.S. Army’s network to get the list of informers, while B’s job is to stop A. So B must learn of the bugs before A does.

A can only do 1000 hours of testing a year. B has full Vista source code, dozens of PhDs, control of the commercial evaluation labs, an inside track on CERT, an information sharing deal with other UKUSA member states, and also runs the government’s scheme to send round consultants to critical industries such as power and telecomms. So B does 10,000,000 hours a year of testing.

After a year, A finds a bug, while B has found 10,000. But the probability that B has found A’s bug is only 1%. Even if B drafts 50,000 computer science graduates to trawl through the Windows source code, he’ll still only get 100,000,000 hours of testing a year. After ten years he will find A’s bug. But by then A will have found nine more.

Even a very moderately resourced attacker can break anything that’s large and complex. There is nothing that can be done to stop this

evaluation

Evaluation is the process of:

`assembling evidence that a system meets, or fails to meet, a prescribed assurance target'.

It is convenient to break this into two cases.

1. the evaluation is performed by the relying party;
2. the evaluation is done by someone other party.

The insurance industry operates laboratories where device tests are conducted. These might involve a fixed budget of effort (perhaps one person for two weeks, or a cost of $15,000). The evaluator starts off with a fairly clear idea of what the device should and should not do, spends the budgeted amount of effort looking for flaws and writes a report. The laboratory then either approves the device, turns it down or demands some changes.

The main failure mode of this process is that it doesn't respond well to progress in attack technology. Eg, for a high-security locks, a lab may demand ten minutes' resistance to picking and say nothing about bumping. Yet bumping tools might improve enough to become a major threat, and picks might get better too.

E1: the orange book

Evaluation of computer security products proposed for government use were conducted according to the ***Orange Book*** — the ***Trusted Computer Systems Evaluation Criteria***. It goes by levels

C1: discretionary access control by groups of users. In fact, this is considered equal to no protection.

C2: discretionary access control by single users; object reuse; audit. It corresponds to carefully configured commercial systems; C2 evaluations were given to IBM mainframe operating systems, and to Windows NT.

B1: mandatory access control — all objects carry security labels and the security policy (BLP or a variant) is enforced independently of user actions. Labelling is enforced for all input information.

B2: structured protection — as B1 plus a *formal model* of the security policy that has been proved consistent with security axioms. Tools must be provided for system administration and configuration management. The TCB must be properly structured and its interface clearly defined. Covert channel analysis must be performed. A trusted path must be provided from the user to the TCB. Severe testing, including penetration testing, must be carried out.

B3: security domains — as B2 but the TCB must be minimal, it must mediate all access requests, it must be tamper-resistant, and it must withstand formal analysis and testing. There must be real-time monitoring and alerting mechanisms, and structured techniques must be used in implementation.

E1: the orange book (ctd)

A1: verification design. As B3, but formal techniques must be used to prove the equivalence between the TCB specification and the security policy model.

The business model of Orange Book evaluations followed traditional government work practices. A government user would want some product evaluated; the NSA would allocate people to do it; given traditional civil service caution and delay, this could take two or three years; the product, if successful, would join the evaluated products list; and the bill was picked up by the taxpayer.

Because of the time the process took, evaluated products were usually one or two generations behind current commercial products. This meant that evaluated products were just not acceptable in commercial markets. The defense computing market stayed small, and prices stayed high.

E1: the rest

The FIPS 140-1 scheme for assessing the tamper-resistance of cryptographic processors; it uses a number of independent laboratories as contractors.

Contractors are also used for *Independent Verification and Validation* (IV&V), a scheme set up by the Department of Energy for systems to be used in nuclear weapons, and later adopted by NASA for manned space flight. In IV&V, there is a simple evaluation target — zero defects.

The Canadians had the *Canadian Trusted Products Evaluation Criteria* (CTPEC) while a number of European countries developed the *Information Technology Security Evaluation Criteria* (ITSEC).

The idea of the latter was that a shared evaluation scheme would help European defense contractors compete against U.S. suppliers with their larger economies of scale. However, ITSEC introduced a pernicious innovation — that the evaluation was not paid for by the government but by the vendor seeking an evaluation on its product.

This motivated the vendor to shop around for the evaluation contractor who would give his product the easiest ride, whether by asking fewer questions, charging less money, taking the least time, or all of the above.

E2: common criteria for itse

The Defense Department began to realise that the Orange Book wasn't making procurement any easier, and contractors detested having to obtain separate evaluations for their products in the USA, Canada and Europe. The mood was in favour of radical cost-cutting

Enter the Common Criteria for Information Technology Security Evaluation

As with ITSEC, evaluations at all but the highest levels are done by CLEFs (commercial licensed evaluation facilities) and are supposed to be recognised in all participating countries, and vendors pay for the evaluations.

The Common Criteria have much more flexibility than the Orange Book. Rather than expecting all systems to conform to BLP, a product is evaluated against a *protection profile*.

There are protection profiles for operating systems, access control systems, boundary control devices, intrusion detection systems, smart-cards, key management systems, VPN clients, and even waste-bin identification systems anyone can propose a protection profile and have it evaluated by the lab of his choice, and so there are a lot of profiles.

It's not that Department of Defense has abandoned multilevel security, so much as tried to develop a more flexible system and so get commercial IT vendors to use the system for other purposes too, and thus defeat the perverse incentives described above.

E2: common criteria for itse (ctd)

The product under test is the *target of evaluation* (TOE).

The rigor with which the examination is carried out is the *evaluation assurance level* (EAL) and can range from EAL1, for which *functional testing* is sufficient, all the way up to EAL7 for which *thorough testing* and *formally verified design* are required. The highest evaluation level commonly obtained for commercial products is EAL4.

A protection profile consists of security requirements, their rationale, and an EAL. It's supposed to be expressed in an implementation-independent way to enable comparable evaluations across products and versions.

A *security target* (ST) is a refinement of a protection profile for a given target of evaluation.

The idea is to first create a protection profile and evaluate it (if a suitable one doesn't exist already), then do the same for the security target, then finally evaluate the actual product.

The first question you should ask when told that some product has a Common Criteria Evaluation is: '*against what protection profile?*' It's not the nominal EAL that tells you anything, but the details of the protection profile.

E2: common criteria (criticisms)

The Common Criteria can sometimes be useful as they give you an extensive list of things to check, and the framework can also help in keeping track of all the various threats and ensuring that they're all dealt with systematically

What people often do, though, is to find a protection profile that seems reasonably close to what they're trying to do and just use it as a checklist without further thought.

It is instead important to understand the Criteria's limitations. They don't deal with

- administrative security measures,
- 'technical/physical' aspects
- crypto algorithms
- evaluation methodology

There is no requirement for evidence that a protection profile corresponds to the real world. So, they avoid all the hard and interesting bits of security engineering.

The most common specific criticism (apart from cost and bureaucracy) is that the Criteria are too focused on the technical aspects of design: things like usability are almost ignored, and the interaction of a firm's administrative procedures with the technical controls is glossed over as outside the scope.

E2: common criteria (criticisms)

Another common criticism is that the Criteria don't cope well with change. Operating systems such as Windows and Linux have been evaluated, but in very restricted configurations. Products with updates, such as Windows with its monthly security patches, are outside the scope.

When presented with a security product, you must always consider whether the salesman is lying or mistaken, and how. The Common Criteria were supposed to fix this problem, but they don't.

So, when presented with an evaluated product, demand what vulnerabilities have been reported or discovered since the evaluation. Look hard at the protection profile: check whether it maps to what you really need. Look how it was manipulated and by whom; whether the CLEF that evaluated the profile was dishonest or incompetent; and what pressure from which government might have been applied behind the scenes.

A really hard-bitten cynic might point out that since the collapse of the Soviet Union, the agencies justify their existence by economic espionage, and the Common Criteria signatory countries provide most of the interesting targets. A false U.S. evaluation of a product which is sold worldwide may compromise 250 million Americans, but as it will also compromise 400 million Europeans the balance of advantage lies in deception.

the mythical man month

The classic book ‘The Mythical Man Month’ argues compellingly that there is no ‘silver bullet’ to solve the problems of software projects that run late and over budget. Exactly the same applies to the problem of assurance and, especially, evaluation.

When you really want a protection property to hold it is vital that the design be subjected to hostile review.

The classic ways of doing hostile review are contractual and conflictual.

An example of the contractual was the Independent Validation and Verification (IV&V) program used by NASA for manned space flight; contractors were hired to trawl through the code and paid a bonus for every bug they found.

An example of the conflictual approach was in the evaluation of nuclear command and control, where Sandia National Laboratories and the NSA vied to find bugs in each others’ designs.

One way of combining the two is simply to hire multiple experts from different consultancy firms or universities, and give the repeat business to whoever most noticeably finds bugs and improves the design. Another is to have multiple different accreditation bodies

and always bear in mind that…

… the interests of the various stakeholders in a system can diverge quite radically.

1. The vendor would prefer that bugs weren't found, to spare the expense of patching.
2. The average customer might prefer the same; lazy customers often don't patch, and get infected as a result.
3. The typical security researcher wants a responsible means of disclosing his discoveries, so he can give the vendors a reasonable period of time to ship a patch before he ships his conference paper; so he will typically send a report to a local *computer emergency response team* (CERT) which in turn will notify the vendor and publish the vulnerability after 45 days.
4. The intelligence agencies want to learn of vulnerabilities quickly, so that they can be exploited until a patch is shipped. (Many CERTs are funded by the agencies and have cleared personnel.)
5. Some hackers disclose vulnerabilities on mailing lists such as bugtraq which don't impose a delay; this can force software vendors to ship emergency patches out of the usual cycle.
6. The security software companies benefit from the existence of unpatched vulnerabilities in that they can use their firewalls to filter for attacks using them, and the anti-virus software on their customers' PCs can often try to intercept such attacks too.
7. Large companies and government depts don't like emergency patches, as it is expensive to test and roll out a new patch against the enterprise's critical systems.