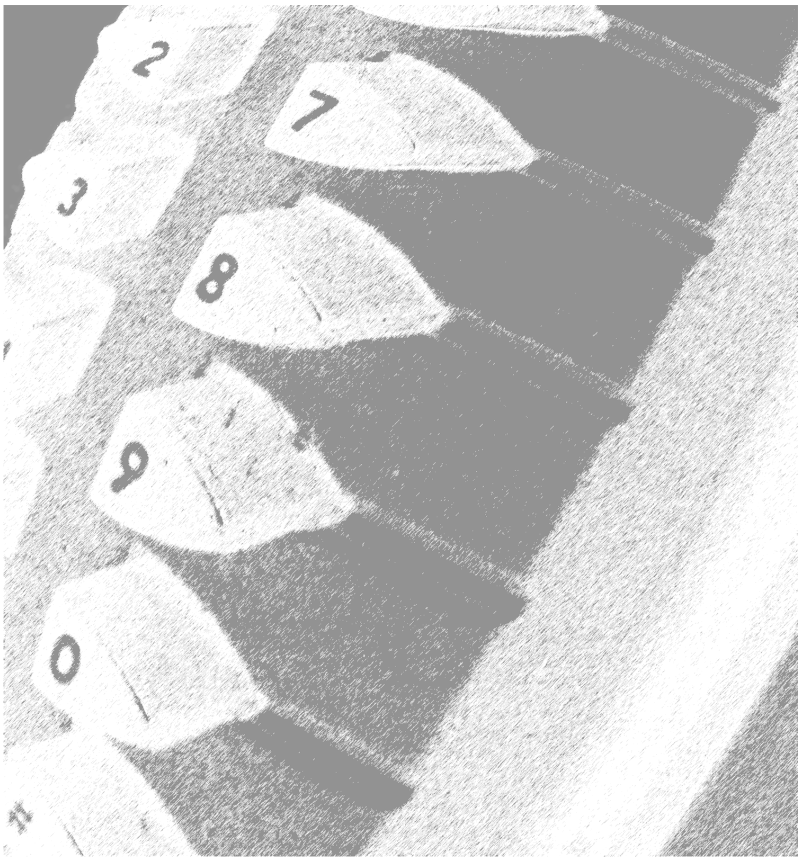
Keeping It Safe 14



14.1 Foreword

In this final chapter of From Architectures to Operating Systems, we will explore the implications of computer security, with a particular focus upon the operating system roles and responsibilities, and reflecting upon what we have encountered in earlier chapters where relevant.

We will introduce some basic concepts and issues in computer security, identify some of the ways in which security threats impact upon computer systems, and also highlight how operating systems and their components can prevent some of these issues from arising or reduce their severity.

Of course some problems happen entirely by misadventure, and there are also issues that can arise due to entirely innocent occurrences. We will also consider some of these and look at mitigating strategies.

14.2 Security threats

Security threats to computer systems can only happen when an unsafe entity is introduced to the system. If a computer system is never connected to the outside world, or any device that might be encountered (such as a flash drive)[185] , and only ever has trusted software installed, then security issues driven by exploitation of technology loopholes should be very rare, if not impossible.

[185] It is claimed that damage to an Iranian centrifuge facility, where uranium is enriched for nuclear projects, was compromised by an infected flash drive in 2010, by a virus known as stuxnet. This highlights the danger of employees using unauthorised devices at work when systems do not have suitable security blocks.

However, as soon as a network is introduced into a computer system or users are able to introduce removable storage media, then a security risk emerges.

Security threats can take many forms:

- Data theft,
- Identity theft,
- Deliberate corruption of data,
- Denial of access to data,
- Denial of access to a service or system,
- Misappropriation of computing resources,
- Imitation
- and, and, and ...

Therefore, there are very many possible ways in which security can be challenged in a modern computer system, and even this set of categories can be expanded into hundreds of variations on those themes. We will examine this spectrum of threats in the following sections.

14.3 Viruses

Viruses are programs that are written to maliciously access or alter the behaviour of a computer system. As they are programs, then they must be executed in order for them to do damage. Once they are running they could achieve any of the classes of threat mentioned above as part of their purpose.

Commencing execution of a virus typically happens when a user is tricked into running a program, or when, for example, an application such as a web-browser has a security flaw that allows a program to run without the user even being asked permission. There are other means, for example an infected boot sector of a bootable device can execute virus code without the user even being aware or able to intervene. This is known as a **boot-sector virus**.

Closing off some of these opportunities will help to reduce the threat. For example, disabling the ability of a machine to boot from any device other than its own primary hard disk unit, will ensure that boot-sector viruses cannot be activated via a temporary storage unit left plugged into the computer. These settings are usually available in the BIOS configuration menu of the system in question. However, this still leaves the possibility of such software arriving via the internet during the machine's everyday operation by a user.

14.3.1 Virus checkers

Virus Checkers (anti-virus software) are programs that potentially inspect every single piece of information being transferred via relevant IO devices (those with a connection to the real world via networks or plug-and-play devices). By matching received data against a set of known data patterns (virus signatures), the virus checker attempts to spot viruses before they can start running. This is done in as streamlined a way as possible. The entire data is not checked byte for byte, rather particular patterns that match known viruses are checked for. This allows the laborious task to be speeded up, though it is certainly a drag on data transfer rates even given this optimisation.

Whilst this model works fairly well, a key problem is that new viruses are being written every day, and therefore virus checkers must be kept up to date with high frequency. Some viruses, known as polymorphic viruses, even use self-created mutations of their own code structure to create duplicates that are not identical, making virus checking via known signatures more difficult.

Once a virus is running, it can do many nasty things. It can simply corrupt the data on the machine, causing serious problems for the user, but there are far more sophisticated things that may be done. Here are a few possibilities you might encounter if you are unlucky:

- It could corrupt file content (as mentioned),
- Encrypt your data and demand payment for its release (**ransomware**),
- It could simply sit quietly in the background and log keystrokes and attempt to steal passwords and credit card numbers (**key-loggers**).
- It could take control of cameras and microphones and record things without the user knowing.
- It could relay data files to a remote server (**data theft**)
- It could use the infected computer to send **spam** emails,
- and more ...

This can be annoying at the very least but potentially catastrophic: imagine an airline whose booking system has been compromised by a keylogger for a month without anyone noticing and all of their booking agents have been typing in the credit card numbers of their customers for that whole time. The customers could be defrauded, and the reputational damage to the airline, or even legal damages claims from customers, might be severe for that company.

14.3.2 Virus examples

According to a leading virus checker software vendor*, these (plus one or two others) are some of the most notable viruses that have been observed since the threat was fully recognised:

CryptoLocker: a ransomware virus that encrypts files on a system making them inaccessible. Files cannot be accessed without paying a sum of money to obtain the decryption key.

ILOVEYOU: spread via email attachments, but once activated, overwrites operating system files as well as corrupting user data.

MyDoom: Among other things, MyDoom took control of computers (usually without the user being aware for some time), and then used

* Sourced from NORTON "The 8 Most Famous Computer Viruses of All Time", https://uk.norton.com/norton-blog/2016/02/the_8_most_famousco.html

those computers to send excessive internet traffic to particular servers, taking them offline for normal users. (a **denial of service** attack).

Storm Worm: also spread by email, resulting in machines being taken over and used to send spam emails, including further spreading of the virus. When a computer is taken over in this way it is known as a zombie or bot infection. Storm Worm was a particularly widespread virus that gained user interaction via an email with the subject line of '230 dead as storm batters Europe'. When curious users opened the email, the potential for the virus to become active was initiated.

Sasser: This virus spreads by scanning remote computer network ports, and attempts to exploit security flaws in operating systems, that allow it to trick the computer into downloading the virus and then doing the same to other machines. Whilst the virus does not do anything 'nasty' to the infected machine directly, it uses CPU and network resources, sometimes to the extent that a companies infrastructure can be disrupted.

Stuxnet: allegedly designed by a national security agency as a cyber-attack against Iranian nuclear weapons development facilities. Iranian centrifuges used for uranium enrichment were targeted by messing up their operating parameters, causing equipment failures. It is suggested that this was transferred to the factory via an infected flash-drive whose user was unaware of its content.

14.4 Architectural exploits

The idea that a well-informed malcontent programmer could utilise their knowledge of the underlying hardware, and perhaps elements of the operating system, in order to exploit a security risk, has existed for a very long time. These are generically known as **side-channel attacks**. However, the sophistication of modern processor technology has released a new wave of opportunities for security loopholes to be discovered and exploited.

The **TLBleed** case is a particularly good example, using side-channel attacks to exploit the relationship between virtual memory and hardware and a component known as the **Translation Look-Aside Buffer** (TLB). TLBleed was developed in 2018, primarily to demonstrate that the flaw existed rather than for malicious purposes. This highlights the complexity of modern systems and how various components interact in ways that are very hard to evaluate fully for security loopholes.

The **Spectre** exploit is another good example. This makes use of branch-prediction and the effects this has upon the timing of certain subsequent behaviours of the machine, during speculative execution, in order to deduce useful information that should be entirely private to another task or process. Remember that processes are meant to be entirely self-contained and unobservable to each other. When any information can leak from one process to another, it can then conceivably be exploited for malicious purposes.

Even features of the hardware system such as DMA can be exploited. **DMA side-channel attack** is a method by which a device connected to a system can initiate DMA data transfers, and because the operating system is not involved, it cannot administer the normal memory privacy protections that apply at the software level. If a computer is not properly protected against this attack then a malicious person could plug in something when the machine is unattended, and access all of the computer's memory content, even if the machine is password locked. BIOS settings may help to prevent such activities from being initiated by plug-in devices.

Most recently, the role of hyperthreading in the mix of other issues that relate to these highly complex architectural exploits has led to hyperthreading being disabled on systems without suitable operating system fixes to alleviate these problems.[186] The sheer complexity of architectures, is, in some ways, making absolute security of a system more difficult to verify, but also offering more and more exotic ways in which hackers can invade the integrity of our computer systems. It seems that there is no likelihood of this kind of problem disappearing anytime soon.

[186] This is more than just an inconvenience. The disabling of key CPU performance features can cause system performance to drop significantly, which could be a serious issue for digital service provision of all kinds.

14.5 Firewalls

Firewalls are filters built into key network infrastructures and/or operating systems that block certain types of data and limit the access to particular resources on the network to particular groups of recognised IP addresses, or perhaps deny access to particular addresses whilst allowing everything else. In theory a firewall can prevent a system from being accessed by a third party trying to access the network from an external connection. This prevents data theft and malicious attacks among other things. Some of these functionalities are often integrated into routers, whilst operating systems can have some of this functionality as standard.

Service attacks, particularly **Denial of Service Attacks**, are malicious behaviours that attempt to overload a server or network connection, such that normal users of the service cannot get their usual services. An online ordering system could be overwhelmed by randomly generated requests that confuse the system and overload the server, meaning that genuine customers cannot place orders because the system is going too slow or is just failing altogether. If this happens to an in-house server, there is not much that can be done except to try to block the offending IP addresses via a firewall so that they do not get to use any system resources by presenting fake requests, but automated attacks can quickly switch to other IP addresses, and therefore a determined attack is hard to mitigate.

With Cloud Computing services, the service providers are more geared up to preventing such attacks, and can switch in extra servers to meet the demand for example, to reduce impact whilst a better solution is found. Specialist software can detect such attacks early and try to automate the IP blocking processes. This is beyond the ability of most small companies, strengthening the case for the value of cloud computing in terms of resilience.

14.6 Encryption and validation

Although data theft is only one part of the security threat spectrum, it is a serious one. This is addressed as far as possible by using encryption. When data is sent via a web-browser, it can be encrypted to a level that most malicious agents would not be able to decode within a reasonable amount of time. It is not a question of *if* an algorithm can be broken, but rather: is the time required and the data obtained really worth the effort? There are multiple encoding/encryption standards in wide use currently, a few common ones are listed here:

AES: Advanced Encryption Standard
RSA: Rivest–Shamir–Adleman algorithm
DES: Data Encryption Standard
SHA: Secure Hash Algorithm[187].

There are also many variations that use aspects of these encryption algorithms, adapting the main core standards that the most widely used encryption concepts are based upon.

Currently there is no cryptography method that is entirely secure, and which can also be used widely in computer systems. The best hope here

[187] Secure hashing, in its own right, is not necessarily able to deliver encryption, though it is often used alongside it. It might be better described as an encoding that allows data integrity to be validated for the sake of resilience or trust, without the data being able to be recoverable from the SHA itself.

is **quantum cryptography**, but this is nowhere near being deployable for everyday computing uses at the present time.

Because computer processing power is increasing year by year, encryption standards theoretically become weaker over time, as it is easier for an encryption scheme to be cracked with a less expensive computer (or more of them). Therefore, the encryption standards also increase in strength over time. A 64 bit encryption standard is weak compared to a 128 or 256 bit standard for example. Additionally, new nuances and variations are introduced to strengthen existing algorithms. The **3DES** standard is much more secure than **DES** for example.

However, this is also a double-edged sword. The need for more and more complex encryption also demands more and more CPU effort. Encryption is a burden on any system, and is unfortunately part of the cost and delay involved in sending and receiving data via a link that is capable of being observed by a third party. On top of which we also have to suffer the impact of constant virus checking efforts. In order to reduce this burden, modern processors may well have instruction set features and possibly specialised hardware units to accelerate encryption and take the burden away from the main processor. The downside is that some of these resources could also be exploited to accelerate attempts to break encryptions on a hacker's machine.

14.7 Data Resilience

A key aspect of computer systems, communications, the internet, and data storage, is the concept of data resilience. In particular we are concerned with the idea that errors at the bit-level can be detected and, ideally, corrected. We assume that technology deals with errors and consequently we rarely see these in the everyday use of our computer systems.

Whilst errors can be very rare, the reality is that no electronic circuit, wifi link, or storage device, is error free, and in some situations such as industrial computing or harsh environments such as space technology, nuclear facilities, and so on, errors are common. For example, in space, cosmic rays can pass randomly through memory circuits in a satellite on-board computer, causing bits to flip state[188]. If that simply causes a very momentary glitch in an audio conversation on a satellite phone link, then this is no big deal. If, on the other hand, that single bit relates

[188] This may seem like a problem exclusively for space systems engineers, and not everyday computing, but even at the altitudes of commercial jets, the rates of bit flip events increases rapidly (of the order of several hundredfold).

to a multi-million dollar digital stock trading transaction, or a guidance system command, then it could be very serious.

Nonetheless, error detection and correction come at a cost, and the more resilient one wishes a system to be, the more effort is required. There are of course many ways to make systems more resilient. One method is to hand-shake important data exchanges. So for example, when a signal is sent, it is sent back and the original sender checks that the signal is still the same. Very much like reading back a delivery address when a customer orders a take-away meal. Not foolproof, but likely to greatly reduce the likelihood of an error. However, it is not always feasible to do this, and methods that allow the data receiver to detect errors independently of the data supplier are important. Some of the best known methods include parity checks and CRC (Cyclic Redundancy Checks), and often such schemes are built into hardware directly, such as the case of ECC error detection in DRAM SIMM and DIMM modules.

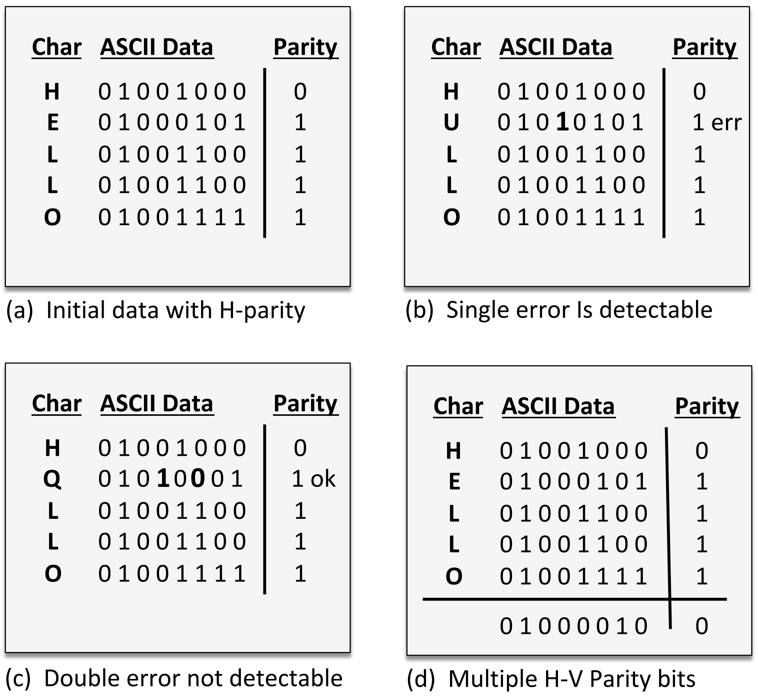
CRC and parity checking methods can also be applied in many contexts, including data storage, data transmission, and memory resilience, so they are not only found in relevant layers of the OSI reference model for example, but are built into the hardware of disk units and memory systems.

So what is parity? And how can it help to detect errors? Let us consider the example of parity checking as given in Figure 14.1. We can see that the simplest method is to apply a single parity bit to each byte.[189] In Figure 14.1(a) we see the original data, with parity bits assigned to each row such that the total number of binary '1' digits in a row is even (known as **even parity**). The parity bit is set to zero or one in order to even up the count in each row. The data is then transmitted or stored along with the calculated parity bits.

[189] In this example we have one parity bit assigned for every 8 bits of data, but we could have any number of bits associated with a parity bit. More data bits per parity bit gives less resilience, as there are more ways an error can occur that one parity bit cannot detect.

In Figure 14.1(b) we see that if one bit is corrupted, then the parity bit no longer agrees with the even parity in that row and the receiver/reader of the data can detect this. This is the core principle of parity checking. However, it might be noted that if there are two errors in the same row, it is possible that they cancel each other out (Bit 'A' flips to a one and Bit 'B' flips to zero, for example). As Figure 14.1(c) shows, this can lead to the error not being detected. What we can say is our **horizontal parity check** is resilient for single bit errors, but not for all possible error cases[190]. Of course single bit errors are typically far more likely than multiple bit errors, so we are at least addressing the most likely case.

[190] You might also have considered here that the parity bit itself could be corrupted. As a single bit error event that would still be detectable.



Char	ASCII Data	Parity
H	0 1 0 0 1 0 0 0	0
E	0 1 0 0 0 1 0 1	1
L	0 1 0 0 1 1 0 0	1
L	0 1 0 0 1 1 0 0	1
O	0 1 0 0 1 1 1 1	1

(a) Initial data with H-parity

Char	ASCII Data	Parity
H	0 1 0 0 1 0 0 0	0
U	0 1 0 **1** 0 1 0 1	1 err
L	0 1 0 0 1 1 0 0	1
L	0 1 0 0 1 1 0 0	1
O	0 1 0 0 1 1 1 1	1

(b) Single error Is detectable

Char	ASCII Data	Parity
H	0 1 0 0 1 0 0 0	0
Q	0 1 0 **1** 0 **0** 0 1	1 ok
L	0 1 0 0 1 1 0 0	1
L	0 1 0 0 1 1 0 0	1
O	0 1 0 0 1 1 1 1	1

(c) Double error not detectable

Char	ASCII Data	Parity
H	0 1 0 0 1 0 0 0	0
E	0 1 0 0 0 1 0 1	1
L	0 1 0 0 1 1 0 0	1
L	0 1 0 0 1 1 0 0	1
O	0 1 0 0 1 1 1 1	1
	0 1 0 0 0 0 1 0	0

(d) Multiple H-V Parity bits

Figure 14.1: Parity checking Examples of use of parity to detect errors.

In Figure 14.1(d) we see that the parity system has been extended to include both **horizontal** and **vertical** parity bits. Typically this is applied to a whole data block of interest - e.g. a payload in a network data packet, or a disk sector, perhaps. You should be able to see that very few error combinations are undetectable. Repeating the error case from Figure (c) leads to vertical parity bits highlighting the errors for instance. Actually, it requires a minimum of 4 bit errors occurring in the same two rows and overlapping columns to create an error case that may be undetectable, and of course this has a very small probability compared to single bit errors, or multiple bits in a single row.

Of course we can make parity error checking more complex still, but often cyclic redundancy checking is used instead (this is a topic you may wish to research yourself).

Ultimately it is good to remember that no system is 100% error resilient, and there is always a cost. In our example message, there are 40 data bits. But with dual parity checking we have a total of 54 bits in this example. This means we have to transmit or manage around 35% more bits and thus get a corresponding reduction in data transfer or storage capacity.

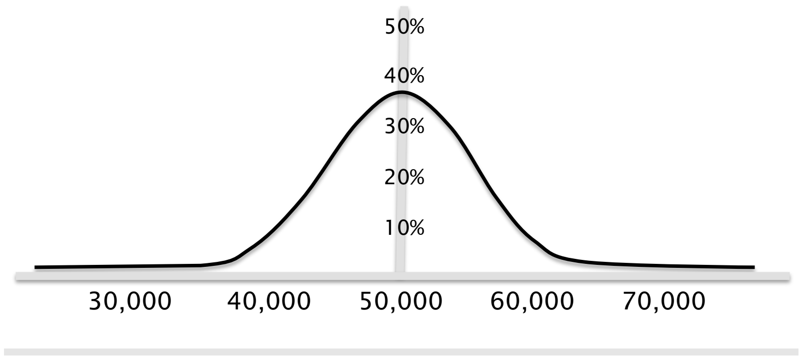


Figure 14.2: Gaussian distribution of some arbitrary hard-disk operating times before failure. The average MTBF is 50,000 hours, with a varying proportion of drives failing at lesser or greater timescales.

The trade-off between having more data bits per parity bit will determine how 'dense' the information is (i.e. transfer or storage capacity) versus how susceptible that data is to bit errors. This has to then be balanced against the severity of impact of a bit error remaining undetected - a matter that is very application dependent.

14.8 Resilient systems

Resilience comes in many forms, and some of these require complex issues for system designers to deal with. All system components can fail. The **Mean-Time-Between-Failure** or **MTBF** is a statistically derived figure that gives us an indication how likely this failure is.

If an HDD manufacturer rates their disks as having an MTBF of 50,000 hours of operation, then this is an average, not a guarantee. Given three drives, one may fail after 40,000 hours, one may fail after 50,000 hours, and one may still be working after 60,000 hours. In actual fact, with the quality of manufacturing, the vast majority of drives will operate within narrow band around the average, and this looks something like Figure 14.2.

A probabilistic curve carries on in both directions indefinitely in theory. It should be possible for example to find that the proportion of HDD units failing after 10,000 hours is, let us say 1%, for a given product. Therefore, in a system that has only one HDD of that type, there is a 1% probability that the system will fail within 10,000 hours.

Using some mathematics, we can work out the probabilities for various different scenarios:

Probability of at least one failure when n drives are installed

The key statement is **at least one** failure: this is the same probability as one minus the probability of no failures at all. And thus if n=5:

$\mathbf{P}_{\mathbf{NoFailures}} = \mathbf{P}_{\mathbf{works}}{}^{n} = 99\%^{5} = 0.951$ (95.1%)
$\mathbf{P}_{\mathbf{OneOrMore}} = (1 - \mathbf{P}_{\mathbf{NoFailures}}) = 1 - 0.951 = 0.049$ (4.9%)

Redundant pairs

Suppose that each drive is duplicated to provide redundancy, and the system can continue as long as no combined pair fails. This means there are now 10 identical drives, and probability of any (i.e. one or more) failures is:

$\mathbf{P}_{\mathbf{none}} = (0.99^{10}) = 0.904$ (90.4%)
$\mathbf{P}_{\mathbf{AnyFailure}} = (1 - \mathbf{P}_{\mathbf{NoFailures}}) = 0.096$ (9.6%)

But the probability of a drive pair failing or working is:
$\mathbf{P}_{\mathbf{PairFail}} = 1\% \times 1\% = 0.01 \times 0.01 = 0.0001$ (0.01%) .
$\mathbf{P}_{\mathbf{PairWorks}} = 1 - 0.0001 = 0.9999$ (99.99%) .

And then finally, the probability of any pair failing is:
$\mathbf{P}_{\mathbf{NoPairsFail}} = (\mathbf{P}_{\mathbf{PairWorks}})^{5} = 0.9999^{5} = 0.9995$ (99.95%)
$\mathbf{P}_{\mathbf{AnyPairFail}} = 1-(\mathbf{P}_{\mathbf{NoPairsFail}}) = 1 - 0.9995 = 0.0005$ (0.05%)

So what do these scenarios actually tell us in practical terms? First of all, a system made up of various components, each with a failure rate of some percentage per year, period of hours, etc, will have an **independent failure rate** of the individual components. That is, the probability of any one component failing. If we assume that any one failure makes the system faulty as a whole then this is the likelihood that the system will fail: the **critical failure**.

On the other hand, we can reduce the potential for a whole system failure by introducing a degree of redundancy. If we introduced paired redundancy in the drive system, the overall chance of any one component failure increases (because there are more components), but the **critical failure**, which is now the possibility that **two** paired drives fail in the same period, is much lower[191], and it is only the latter case that causes a whole system failure.

[191] Indeed, in our example cases, we found that the likelihood of a critical failure was reduced by a factor of about 100.

Furthermore, once redundancy is introduced, an engineer could immediately be alerted to the first drive failure in a pair and then very quickly replace it, and then we will never encounter a double failure: the possibility of a second failure within a few hours of the first, in the same pair, and the first being yet to be replaced, is minuscule. In this scenario it is arguable that the chance of critical failure is as good as zero in practice.[192]

[192] However, this doesn't mean it should be ignored. In a large installation, such as a data-centre, we still need to predict failure rates to ensure enough engineers are available to do the work and that enough spare components are in stock. So we move from predicting failure to predicting maintenance demands.

The redundancy effect is the basis of many fault tolerant systems, and indeed some RAID disk array configurations use exactly this principle. If the drive modules are hot-swap compatible[193], then the engineer's work will be unnoticed, and nobody will ever be aware of the problem.

[193] Hot-Swap components are components that may be plugged in and removed without halting or powering down the host system.

14.9 UPS systems

Resilience comes in many forms, and some of these are quite complex issues for system designers to deal with. One obvious issue that is often overlooked is the simple fact that without electricity, a system will fail catastrophically within moments, unless suitable mechanisms are put in place to deal with this.

The obvious thing to do here is to ensure that power is not interrupted at all. However, in practice, this is not likely to be feasible for most situations. The best that can be expected is to provide a short term reserve of power to be used for a few minutes whilst the system reacts to the situation, and follows a remediation procedure. Typically a system would aim to flush any uncommitted data held in memory back to disk, close any files, and stop any critical processes, to prevent data loss, and in the event of a safety critical system also place the system into as safe a mode as it is able to be designed to do.

For the general user, the Uninterruptible Power Supply (UPS) is a solution that allows a modest degree of resilience. These systems can cost a few hundred pounds for a small system, and may permit a computer system or small server to run for a short period of perhaps minutes. When the UPS detects a power outage, it sends a message to the computer to tell it that it must follow its shut down procedure. In professional enterprise systems and server farms, UPS systems can be extremely sophisticated, and needless to say, very expensive. They can potentially ensure that limited critical systems continue to operate for hours, whilst less important systems are shut down cleanly.

14.10 Resilience and safety

We have talked already about redundancy in terms of providing duplicate resources to reduce the critical failure rate. However, no system can be fault-free forever. When systems do fail, they may have to do so in a particular way, in order to ensure a safe outcome is reached. Such systems are often encountered where they manage highly critical requirements, and are often referred to as **safety-critical systems**. They can be found in many environments, including industrial, medical, automotive, and aerospace, to name but a few.

An example that is often talked about at the present time is the idea of a self-driving car. There are many obvious safety concerns here, if such a vehicle suffered a technical failure. Let us take the simplest task: ensuring the vehicle remains between two sets of white lines on a road. If this function is determined by a processor running an evaluation algorithm to analyse sensor data, camera feeds, and so on, then this processor is a critical point of failure.

If the processor were to suddenly stop working, then the system would crash: literally! But actually there are other worrying possibilities too. What if the processor continued to operate but started to generate erroneous results: the algorithm would cease to function correctly, and could potentially be even more dangerous than the first case.

In a system like this, a common solution is to introduce redundancy, but in a particular way. For instance, three processors could be installed to independently compute the same algorithm (**triple-redundancy**), and then the three processor outputs would be compared.[194] Ideally all three agree, but if one processor becomes faulty and starts to disagree, the other two will still agree, and they will determine how to steer the car. This situation would also cause the system to move into a safe-mode, perhaps pulling over at the next safe place on the road within a 30-second emergency stop procedure for example.

[194] This is also often referred to as **triple modular redundancy** or **triple-mode redundancy** (TMR).

This type of redundancy offers high degrees of integrity. The probability of two processors spontaneously failing within the space of 30 seconds before the car can safely pull over is vanishingly small. If this actually happened, it might well be due to some single cause: such as fire or electrical shorts in the control unit and not random failures. These are factors that can also be engineered down toward very low probabilities of occurrence.

An important point was encountered here. When the car suffered a fault, we imagined it moving into a safe mode, reaching a position where it can shut down without danger. This is exactly what we wish to happen in such systems, and this is known as **graceful degradation**. If a sensor fails, the car may decide that it just has to reduce its maximum speed. If two sensors fail, it may have to decide to leave the motorway at the next exit to seek assistance. If a processor fault occurs, it may first of all reboot that specific processor and check if things have reverted to normal, if not then the safe stop-and-park mode is invoked. These are all scenarios that depend upon certain failures and their severity being understood and designed for within the whole system design, and which we will increasingly have to come to know and work with in the future world of computers.

14.11 Summary

We have learned that computer systems are vulnerable to many faults, both the kind that occur due to simple failure of equipment and components, but also those more deliberate acts of intent.

One field that never stands still is computer security. In this chapter we explored some basic security issues and ended up exploring some highly complex ideas such as DMA side-channel attack, which could only be understood with a knowledge of hardware that is often overlooked.

In a sense, many of the easy software-based routes to security loopholes have been thoroughly understood, and increasingly closed off as solutions are deployed.

However, the response to this - an increasing emergence of security exploits that use deep architectural knowledge, such as branch-prediction side effects, means that future security experts may well have to know a lot more about hardware than software to be fully prepared.

14.12 Terminology Introduced in this chapter

Denial of service attack	Encryption
Encryption algorithm	Independent failure rate
Key-logger	MTBF
Ransomware	Security threat
Side-channel attack	Virus
Virus checker	

These terms are defined in the glossary, Appendix A.1.