Lectures Notes on Secure and Dependable Systems

Florian Rabe

2017

Contents

Part I

Introduction

Chapter 1

Meta-Remarks

Important stuff that you should read carefully!

State of these notes I constantly work on my lecture notes. Therefore, keep in mind that:

- I am developing these notes in parallel with the lecture—they can grow or change throughout the semester.
- These notes are neither a subset nor a superset of the material discussed in the lecture.
- Unless mentioned otherwise, all material in these notes is exam-relevant (in addition to all material discussed in the lectures).

Collaboration on these notes I am writing these notes using LaTeX and storing them in a git repository on GitHub at https://github.com/florian-rabe/Teaching. Familiarity with LaTeX as well as Git and GitHub is not part of this lecture. But it is essential skill for you. Ask in the lecture if you have difficulty figuring it out on your own.

As an experiment in teaching, I am inviting all of you to collaborate on these lecture notes with me.

By forking and by submitting pull requests for this repository, you can suggest changes to these notes. For example, you are encouraged to:

- Fix typos and other errors.
- Add examples and diagrams that I develop on the board during lectures.
- Add solutions for the homeworks if I did not provide any (of course, I will only integrate solutions after the deadline).
- Add additional examples, exercises, or explanations that you came up or found in other sources. If you use material from other sources (e.g., by copying an diagram from some website), make sure that you have the license to use it and that you acknowledge sources appropriately!

The TAs and I will review and approve or reject the changes. If you make substantial contributions, I will list you as a contributor (i.e., something you can put in your CV).

Any improvement you make will not only help your fellow students, it will also increase your own understanding of the material. Therefore, I can give you up to 10% bonus credit for such contributions. (Make sure your git commits carry a user name that I can connect to you.) Because this is an experiment, I will have to figure out the details along the way.

Other Advice I maintain a list of useful advice for students at https://svn.kwarc.info/repos/frabe/Teaching/general/advice_for_students.pdf. It is mostly targeted at older students who work in individual projects with me (e.g., students who work on their BSc thesis). But much of it is useful for you already now or will become useful soon. So have a look.

Chapter 2

Concepts

This section mostly follows [ALR01], including some tables, which is available online.

Computer systems can be evaluated according to

- functionality
- usability
- performance
- cost
- dependability

Dependability means to deliver service to a user that can justifiably be trusted. The user is another system (physical or human) that interacts with the former at the service interface. The **function** of a system is what the system is intended to do as described by its functional specification. Correct service is delivered when the service implements the specification.

A systematic exposition of the concepts of dependability consists of three parts: the attributes of dependability, their threats, and means to achieve dependability. An overview is given in Fig. 2.1.

2.1 Attributes

2.1.1 Definitions

Availability means the readiness for correct service. This includes two subaspects: the functionality has to **exist** and be realized **correctly**, i.e., according to its specification.

Reliability means the continuity of correct service. This is similar to availability but emphasizes that the service not only exists but exist without downtime or intermittent failures.

Safety means the absence of catastrophic consequences on the user(s) and the environment. Often safety involves interpreting signals received from and sending signals to external devices that operate in the real world, e.g., the cameras and the engine of the car. This introduces additional uncertainty (not to mention the other cars and pedestrians) that can be difficult to anticipate in the specification.

Security means the availability for authorized users only. This includes protection against any malicious influenced from the outside, i.e., any kind of attack or hacking. This includes all defenses against hacking.[1]

Confidentiality means the absence of unauthorized disclosure of information. This includes the security of all private data including any intermediate results of computation such as passwords or keys.

Maintainability means the ability to undergo repairs and modifications. This includes all necessary changes needed during long-term deployment.

2.1.2 Incomplete Specifications

Often the attributes are not or only implicitly part of the specification of a system.

[1] [ALR01] calls a related property *integrity*, and then defines security as the combination of integrity and confidentiality.

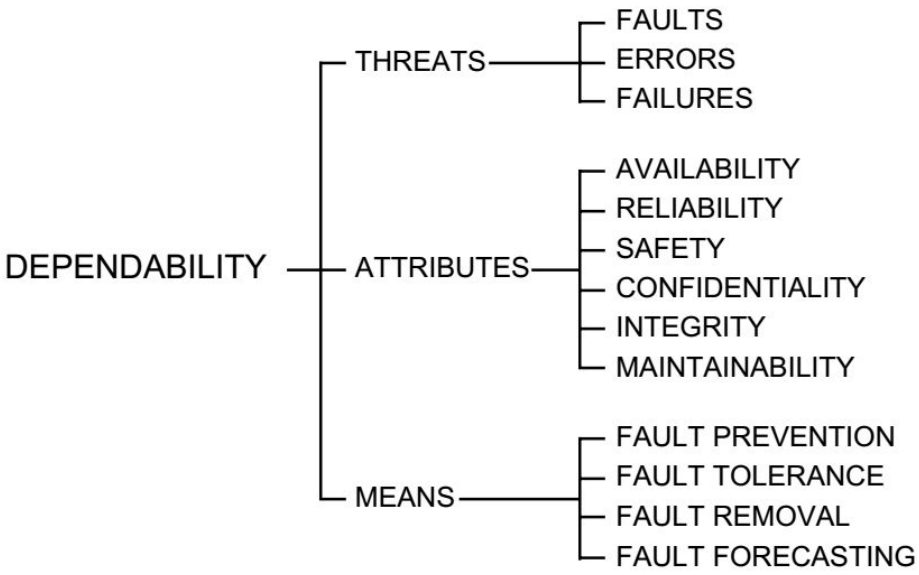

Figure 2.1: Concepts Related to Dependability

Availability is usually implicitly required, but details may be omitted. For example, the acceptable variation of response times may be unspecified.

Reliability is also usually implicit required, but details may be omitted. For example, the acceptable downtime may be unspecified.

Safety is usually specified well if a potential safety danger is realized. But it can be easy to foresee all necessary safety requirements.

Security is often forgotten completely or partially. It is usually difficult to translate the abstract requirement of security into concrete, testable properties. After all, the first mistake of security is to assume to know what the attacker might do.

Confidentiality is often considered even less then security.

Maintainability is often ignored completely. That is a typical pitfall for large projects, where realizing any requirement at deploy time may be completely different from realizing it after a year or later. This is because intermediate changes have messed up the system so much that, e.g., security flaws are not noticed anymore.

2.1.3 Imperfect Implementations

All attributes are very difficult to realize perfectly in implementations.

Availability and reliability often fail. In addition to plain design or implementation errors, there may be failures in hardware, networks, operating system, external components that were not foreseen by the developer.

Together with correctness, safety is the only property that is at least in principle accessible to a formal definition. But the resulting problem is undecidable. So in practice, we have to use extensive testing.

Security is very complex to prove because any proof must make assumptions about what kind of attacks there are. Attacking a system often requires intentionally violating the specification and supply unanticipated input.

Perfect confidentiality is impossible to realize because all computation leaks some information other than the output: This reaches from runtime and resource use to obscure effects like the development of heat due to CPU activity.

Maintainability is hard to realize because especially inexperienced developers or unskilled managers cannot assess whether a particular design is maintainable.

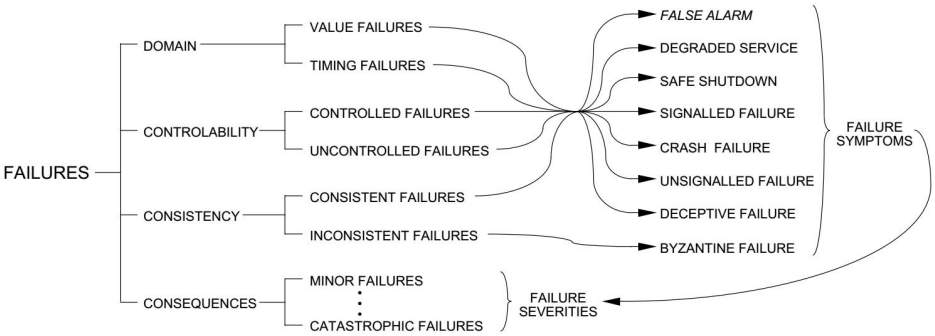

Figure 2.2: Failure Modes

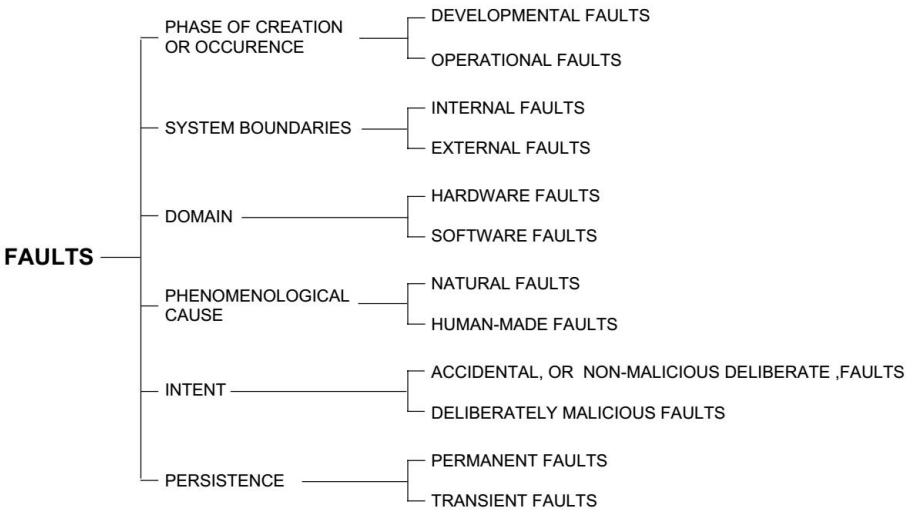

Figure 2.3: Fault Classes

2.2 Threats

2.2.1 Failures

A system **failure** is an event that occurs when the delivered service deviates from correct service. See Fig. 2.2 for an overview of related concepts.

A **fault** is the adjudged or hypothesized cause of an error. A fault is **active** when it produces an error; otherwise it is **dormant**. See Fig. 2.3 for an overview of related concepts.

An **error** is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service.

2.3 Means

2.3.1 Fault Prevention

Fault prevention requires quality control techniques employed during the design and manufacturing of the system. That includes, e.g., structured programming, information hiding, or modularization.

Shielding, radiation hardening, etc., are needed to prevent physical faults. Training and maintenance procedures aim at preventing interaction faults. Firewalls and similar defenses intend to prevent malicious faults.

2.3.2 Fault Tolerance

Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. This may refer to accidental or malicious faults.

Error detection An error that is present but not detected is a **latent** error. Concurrent error detection takes place during service delivery. Preemptive error detection takes place while service delivery is suspended; it checks the system for latent errors and dormant faults.

Recovery Recovery consists of error handling and fault handling.

Error handling eliminates errors from the system state by

- rollback: the state is transformed to a previously saved state
- compensation: the erroneous state is redundant enough to eliminate the error,
- rollforward: the state is transformed to a new state without the detected error.

Fault handling prevents located faults from being activated again by

- fault diagnosis: identify location and type of the cause of an error
- fault isolation: exclude the fault from future service delivery, i.e., make the fault dormant
- system reconfiguration: switch to non-failed components
- system reinitialization: update to a new configuration of the system

The choice of error detection, error handling, and fault handling techniques, and of their implementation, is directly related to the underlying fault assumption.

Fault tolerance is recursive: Its implementation must be protected against faults as well.

2.3.3 Fault Removal

Development Phase Fault removal during the development phase of a system consists of three steps:

- Verification checks whether the system satisfies required properties.
- If verification fails, diagnosis identifies the faults that prevented verification.
- After diagnosis, correction is carried, and verification repeated.

Multiple different properties can be verified:

- Verifying the specification is usually referred to as validation. Static verification does not exercise the systems and uses static analysis (e.g., inspections or walk-through), model-checking, or theorem proving. Dynamic verification exercises the systems and uses testing.
- Verifying the fault tolerance mechanism. This can employ formal static verification or fault injection, i.e., testing where intentional faults or errors are part of the test.
- Verifying that the system cannot do more than specified is especially relevant for safety and security.

Design for verifiability means to design a system in such a way that verification becomes easy.

Operation Phase Fault removal during the life time of a system employs two methods:

- Corrective maintenance removes faults that have produced errors that were detected and reported.
- Preventive maintenance uncovers faults before they cause errors. This may also include design faults that have caused errors in similar systems.

Fault removal during operation often first isolates the fault (e.g., by a workaround or patch) before the actual removal is carried out.

2.3.4 Fault Forecasting

Fault forecasting evaluates a system with respect to fault occurrence or activation. It has two aspects:

- Qualitative evaluation identifies, classifies, and ranks the failure modes, or the event combinations that would lead to system failures.
- Quantitative evaluation determines the probabilities to which some of the attributes of dependability are satisfied, which are then viewed as measures of dependability. This may use, e.g., Markov chains or Petri nets.

Chapter 3

Challenges

This chapter lists examples of disasters and failures that serve as examples of what secure and dependable systems should avoid.

The lists are not complete and may be biased by whether

- I became aware of it and found it interesting enough
- the cause could be determined and was made public

Feel free to edit these notes by adding important examples that I forgot when I compiled the lists.

All damage estimates are relative to the time of the event and not adjusted to inflation.

Note that for security problems, the size of the damage is naturally unknown because attacks will typically remain secret. Only the cost of updating the systems can be estimated, which may or may not be indicative of the severity of the security problem.

3.1 General Aspects

State-of-the-art software and hardware systems simply are not safe, secure, and dependable. Moreover, we do not understand very well yet how to make them so.

This is different from many other areas such as mechanical or chemical engineering. While these occasionally cause disasters, these can usually be traced back to human error, foul play, or negligent or intentional violation of regulations. Such disasters usually result in criminal proceedings, civil litigation, or revision or extension of regulations.

The situation is very different for computer systems. There is no general methodology for designing and operating computer systems well that can be easily described, taught, or codified.

The situation will hopefully improve over the course of the 21st century. The problem has been recognized decades ago, and many companies and researchers are working on it. They approach from very different directions with different goals and different methodologies.

This has resulted in a wide and diverse variety of not coherently connected methods with varying degrees of depth, maturity, cost, benefit, and practical adoption.

A typical effect is a trade-off along a spectrum of methods:

- cheap but weak methods on one end
- strong but expensive methods on the other end.

Therefore, it is often necessary to choose a degree of safety assurance rather than actually guarantee safety. This spectrum is so extreme that

- the majority of practical software development does not systematically ensure any kind of safety,
- the majority of theoretical solutions are neither ready nor affordable for practical use.

Incidentally, this means that this course's subject matter is much less well-defined than that of other courses.[1] That makes it particular difficult to design a syllabus for. It will give an overview of the most important state-of-the-art

[1] For example, the other two courses in this module almost design themselves because the subject matter is very well understood and standardized.

methods.

3.2 Major Disasters Caused by Programming Errors

Space Exploration There have been a number of failures in space exploration due to minor programming errors. These include

- 1962, Mariner 1 rocket lost: misread specification (overlooked bar over a variable) was implemented (damage around $20 million)
- 1982, Viking I lost: software update written to wrong memory area overriding vital parameters for antenna
- 1988, Phobos 1 lost: one character missing in software update led to accidentally executing a testing routing at the wrong time
- 1996, Mars Global Surveyor lost: data written to wrong memory addresses
- 1999, Mars Polar Lander lost: presumably software not accounting for false positive when detecting shutdown even though the possibility was known
- 2004, Mars Rover Spirit lost for 16 days: delay in deleting obsolete files led to lack of available flash memory, which triggered a reboot, which led to a reboot cycle

Therac-25 Between 1985 and 1987, the Therac-25 machine for medical radiation therapy caused death and/or serious injury in at least 6 cases. Patients received a radiation overdose because the high intensity energy beam was administered while using the protection meant for the low intensity beam.

The cause was that the hardware protection was discontinued, relying exclusively on software to prevent a mismatch of beam and protection configuration. But the software had always been buggy due to a systemic failures in the software engineering process including complex systems (code written in assembly, machine had its own OS), lack of software review, insufficient testing (overall system could not be tested), bad documentation (error codes were not documented), and bad user interface (critical safety errors could be manually overridden, thus effectively being warnings).

Details: https://en.wikipedia.org/wiki/Therac-25

Patriot Rounding Error In 1991 during the Gulf war, a US Patriot anti-missile battery failed to track an incoming Iraqi Scud missile resulting the death of 28 people.

The cause was a rounding error in the floating point computation used for analyzing the missile's path. The software had to divide a large integer (number of $0.1s$ clock cycles since boot 100 hours ago) by 10 to obtain the time. This was done using a floating-point multiplication by 0.1—but 0.1 is off by around 0.000000095 when chopped to a 24-bits binary float. The resulting time was off by 0.3 seconds, which combined with the high speed of Scud missile led to a serious miscalculation of the flight path.

Details: http://www-users.math.umn.edu/~arnold/disasters/patriot.html

Ariane 5 In 1996, the first launch of an Ariane 5 rocket (at a cost of over $300 million for rocket and payload) failed, and the rocket had to be destructed after launch. Both the primary and the backup system had shut down, each trying to transfer control to the other, after encountering the same behavior, which they interpreted as a hardware error.

The cause was an overflow exception in the alignment system caused by converting a 64-bit float to a 16-bit integer, which was not caught and resulted in the display of diagnostic data that the autopilot could not interpret. The programmers were aware of the problem but had falsely concluded that no conversion check was needed (and therefore omitted the check to speed up processing). Their conclusion had been made based on Ariane 4 flight data that turned out to be inappropriate for Ariane 5.

The faulty component was not even needed for flight and was only kept active for a brief time after launch for convenience and in order to avoid changing a running system.

Details: http://www-users.math.umn.edu/~arnold/disasters/ariane5rep.html

Intel Pentium Bug In 1994, it was discovered that the Intel Pentium processor (at the time widely used in desktop computers) wrongly computed certain floating point divisions. The cost of replacing the CPUs was estimated at about $400 million.

The error occurred in about 1 in 9 billion divisions. For example, 4195835.0/3145727.0 yielded 1.333739068902037589 instead of 1.333820449136241000.

The cause was a bug in the design of the floating point unit's circuit.

Kerberos Random Number Generator From 1988 to 1996, the network authentification protocol Kerberos used a mis-designed random number generation algorithm. The resulting keys were so predictable that brute force attacks became trivial although it is unclear if the bug was ever exploited.

The cause was the lack of a truly random seed value for the algorithm. Moreover, the error persisted across attempted fixes because of process failures (code hard to read, programmers had moved on to next version).

Detail: http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2331&context=cstech

USS Yorktown In 1997, critical navigation and weapons hardware on the USS Yorktown was paralyzed at sea for 3 hours while rebooting machines.

The cause was a blank field in a database that was interpreted as 0 leading to a division-by-zero. Special floating point values such as infinity or NaN were not used, thus resulting in an exception. The exception was handled by neither the software nor the operating systems (Windows NT) thus crashing both.

Details: http://www.cs.berkeley.edu/~wkahan/Boulder.pdf

Mars Climate Orbiter In 1998 the Mars Climate Orbiter was lost causing damage of around $300 million after software had calculated a false trajectory when updating the position of the spacecraft.

The cause was that two components by different manufacturers exchanged physical quantities as plain numbers (i.e., without units). One component assumed customary units (pound seconds) whereas the other assumed SI units (Newton seconds). The first component was in violation of the specification of the interface.

Year 2000 and 2038 Problems Leading to the year 2000, about $300 billion were spent worldwide to update outdated software that was unable to handle dates with a year of 2000 or higher.

The cause was that much software was used far beyond the originally envisioned lifetime. At programming time, especially at times when memory was still scarce, it made sense to use only two digits for the year in a date. That assumption became flawed when dates over 2000 had to be handled.

A related problem is expected in the year 2038. At that point the number of seconds since 1970-01-01, which is the dominant way of storing time on Unix, will exceed the capacity of a 32-bit integer. While application software is expected to be updated by then anyway, modern embedded systems may or may not still be in use.

Los Angeles Airport Network Outage In 2007, LA airport was partially blocked for 10 hours due to a network outage that prevented passenger processing. About 17,000 passengers were affected.

The cause was a single network card malfunction that flooded the network and propagated through the local area network.

Details: https://www.oig.dhs.gov/assets/Mgmt/OIGr_08-58_May08.pdf

Debian OpenSSL Random Number Generator From 2006 to 2008 Debian's variant of OpenSSL used a flawed random number generator. This made the generated keys easily predictable and thus compromised. It is unclear whether this was exploited.

The cause was that two values were used to obtain random input: the process ID and an uninitialized memory field. Uninitialized memory should never be used but is sometimes used as a convenient way to cheaply obtain a random number in a low-level programming language like C. The respective line of code had no immediately obvious purpose because it was not commented. Therefore, it was removed by one contributor after code analysis tools had detected the use of uninitialized memory and flagged it as a potential bug.

Detail: https://github.com/g0tmi1k/debian-ssh

Knight Capital Trading Software In 2012, high-frequency trading company Knight Capital lost about $10 million per minute for 45 minutes trading on the New York Stock Exchange.

The cause was an undisclosed bug in their automatic trading software.

Heartbleed From 2012 to 2014, the OpenSSL library was susceptible to an attack that allowed remotely reading out sections of raw physical memory. The affected sections were random but repeated attacks could piece together large parts of the memory. The compromised memory sections could include arbitrary critical data such as passwords or encryption keys. OpenSSL was used not only by many desktop and server applications but also in portable and embedded devices running Linux. The upgrade costs are very hard to estimate but were put at multiple $100 millions by some experts.

The cause was a bug in the Heartbeat component, which allowed sending a message to the server, which the server echoed back to test if the connection is alive. The server code did not check whether the given message length l was actually the length of the message m. Instead, it always returned l bytes starting from the memory address of m even if l was larger than the length of m. This was possible because the used low-level programming language (C) let the programmers store m in a memory buffer and then over-read from that buffer. Moreover, their C code is so hard to read that it is impossible to notice such minor errors on a cursory inspection.

Details: http://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Shellshock From 1998 to 2014, it was possible for any user to gain root access in the bash shell on Unix-based systems. The upgrade cost is unknown but was generally small because updates were rolled out within 1 week of publication. Moreover, in certain server applications that passed data to bash this was possible for arbitrary clients as well.

The cause was the use of unvalidated strings to represent complex data. Bash allowed storing function definitions as environment variables in order to share function definitions across multiple instances. The content of these environment variables was trusted because function definitions are meant to be side-effect-free. However, users could append $;C$ to the value of an environment variable defining a function. When executing this function definition, bash also executed C.

Independently, many server applications (including the widely used cgi-bin) pass input provided by remote users to bash through environment variables. This resulted in input provided by remote clients being passed to the bash parser, which was against the assumptions of the parser. Indeed, several bugs in the bash parser caused remotely exploitable vulnerabilities.

Details: https://fedoramagazine.org/shellshock-how-does-it-actually-work/

Apple 'goto fail' Bug From 2012 to 2014, Apple's iOS SSL/TLS library falsely accepted faulty certificates. This left most iOS applications susceptible to impersonation or man-in-the-middle attacks. Because Apple updated the software after detecting the bug, its cost is unclear.

The immediately cause was a falsely-duplicated line of code, which ended the verification of the certificate instead of moving on to the next check. But a number of insufficiencies in the code and the software engineering process exacerbated the effect of the small bug.

The code was as follows:

```
static OSStatus SSLVerifySignedServerKeyExchange(
  SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
  uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
```

```
    SSLFreeBuffer(&hashCtx);
    return err;
```

In a better programming language that emphasizes the use of high-level data structures, the bug would likely not have happened or be caught easily. But even using C, it could have been caught by a variety of measures including unreachable code analysis, indentation style analysis, code coverage analysis, unit testing, or coding styles that enforce braces around one-command blocks.

Details: https://www.imperialviolet.org/2014/02/22/applebug.html

3.3 Other Interesting Failures

Odyssey Court Software In an ongoing crisis since 2016, US county court and California and other states have been having difficulties using the new Odyssey software for recording and disseminating court decisions. This has caused dozens of human rights violations due to erroneous arrests or imprisonment. This includes cases where people spent 20 days in jail based on warrants that had already been dismissed.

The cause is a tight staffing situation combined with the switch to a new, more modern software system for recording court decisions. The new software expects uses more high-level data types (e.g., reference to a law instead of string) in many places. This has led to the erroneous recording of decisions and a backlog of converting old decisions into the new database (including decisions that invalidate decisions that are already in the database).

Details: https://arstechnica.com/tech-policy/2016/12/court-software-glitches-result-in-erroneous-arrests-de

Other Failures Caused By System Updates This is a selection of failures that did not cause direct damage but led to availability failures on important infrastructure.

In 1990, all AT&T phone switching centers shut down for 9 hours due to a bug in a software update. An estimated 75 million phone calls were missed.

In 1999, a faulty software update in the British passport office delayed procedures. About half a million passports were issued late.

In 2004, the UK's child support agency EDS introduced a software update while restructuring the personnel. This led to several million people receiving too much or too little money and hundreds of thousands of back-logged cases.

In 2015, the New York Stock Exchange had to pause for 3 hours for a reboot after a software problem. 700,000 trades had to be canceled.

In 2015, hundreds of flights in the North Eastern US had to be canceled or delayed for several hours. The cause was a problem with new and behind-schedule computer system installed in air traffic control centers.

FBI Virtual Case File Project In 2005 the Virtual Case File project of the FBI, which had been developed since 2000, was scrapped. The software was never deployed, but the project resulted in the loss of $170 million of development cost.

The cause was systemic failures in the software engineering process including:

- poor specification, which caused bad design decisions
- repeated specification changes
- repeated change in management
- micromanagement of software developers
- inclusion of many personnel with little training in computer science in key positions

These problems were exacerbated by the planned flash deployment instead of a gradual phasing-in of the new system—a decision that does have advantages but made the systems difficult to test and made it easier for design flaws to creep in. The above had two negative effects on the code base

- increasing code size due to changing specifications
- increasing scope due to continually added features

which exacerbated the management and programming problems.

Fooling Neural Networks Deep learning neural networks have become very popular recently for pattern recognition (pictures and speech in particular). They are already used in, e.g., in self-driving cars or the AlphaGo system.

Despite their economic importance, they can err spectacularly in ways that are entirely different from the ways human pattern recognition errs. This is unavoidable because it is a consequence of their mathematical foundations.

Researchers were able to demonstrate that

- minor changes to an image that would be imperceptible to a human can make the network identity as something entirely different (e.g., a lion as a library)
- images that are unrecognizable to humans can make the network see an object with absolute certainty (e.g., labeling white noise as a lion)

This has major implications for the safety and security of systems based on these networks.

Details: http://www.evolvingai.org/fooling

Excel Gene Names In 2016, researchers found that about 20% of papers in genomics journals contain errors in supplementary spreadsheets.

The cause is that Microsoft Excel by default guesses the type of cell data that is entered as a string and converts the string into that type. This affects gene names like "SEPT2" (Septin 2, converted to the date September 02) or REKIN identifiers like "2310009E13" (converted to the floating point number $2.31E+13$).

Details: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-1044-7

Failures in Involving Computer-Related Manufacturing This is a selection of other notable failures that involve hardware manufacturing.

In 2006, two Airbus plants used incompatible version of CAD software. This resulted in cables being produced too short to connect.

In 2006, Sony batteries mostly used in Dell notebooks had to be recalled. The resulting cost was about $100 million.

In 2016, Samsung Galaxy phones had to be recalled due to faulty batteries.

3.4 Major Vulnerabilities due to Weak Security

3.4.1 Software and Internet

Operating Systems Vulnerabilities in operating systems are dangerous because only a few systems are used worldwide, therefore any problem is shared by many users. Moreover, the operating system usually has full access to the computer and its network, which allows any attack to do great damage.

Moreover, operating systems are usually bundled with standard applications (e.g., web browser, email viewer). These are tightly integrated with the OS (e.g., by using the same libraries for encryption or accessing files). Thus, a vulnerability often badly affects the majority of users who use these standard applications.

In 2000, the ILOVEYOU worm exploited weaknesses in the Windows OS and Outlook mail service, by infecting a significant share of all internet-connected computers within a few days. Its damage was estimated at over $5 billion and the removal costs at over $410 billion.

In recent years operating system companies have reacted to these problems. They have become more sensitive to security issues and allow for coordinated disclosure of vulnerabilities together with swift updates. Most noticeable for end users is the urged tendency to frequently install updates. For example,

- Microsoft Windows 10 automatically downloads and installs updates in a way that users cannot prevent.
- Google's Android now reserves the right to download minor updates immediately, even via mobile data.

This has greatly reduced the frequency of major problems.

An additional problem is that attacks are often conducted by state governments for purposes of terrorism, oppresion, espionage, sabotage, or law enforcement.

In 2010, the stuxnet worm was used by presumably the US and/or Israel to sabotage Iran's nuclear program. It was the most sophisticated attack to become public, involving multiple zero-day exploits and including attacks on programmable logic controllers.

In 2013, Edward Snowden revealed a massive secret surveillance program run by the US government. It used many ways to intercept data sent by users either at transmission nodes or at company data centers. This included connection metadata and any unencrypted or decryptable content. The stolen data remains mostly secret so that it prevents clarity on the information compromised and its usage. In response, many software companies introduced

end-to-end encryption that precludes even themselves to access their users data.

In 2016, Citizen Lab discovered an attack that used previously unknown vulnerabilities in Apple's Safari on iOS. It allowed, for the first time, an attacker to remotely take full control of the iPhone, triggered as soon as Safari was pointed to the attack URL. It is suspected that the exploit was produced commercially by the Israeli company NSO Group and used (at least) by the United Arab Emirates to spy on dissidents.

Details: http://www.vanityfair.com/news/2016/11/how-bill-marczak-spyware-can-control-the-iphone

Cloud Services Consumers are more and more using internet services for their processing needs. These include

- file storage, e.g., via Dropbox
- email and calendar services, e.g., via gmail
- office applications, e.g., directly via Google's office web site or indirectly via Microsoft's office suite
- social networking, e.g., via Facebook

Most modern operating systems and their bundled applications store large amounts of user data on the company's web servers, including, e.g., message archive, photographs, or location history. This creates unprecented risks for privacy, with legal regulation mostly lagging behind. (Most legislation were designed to limit the government from violating privacy. Corporations were barely restricted, in fact they used to have less power.)

Because most users do not understand the technical issues and blindly accept terms of service, thus generously granting access rights to applications, more and more user data becomes available to the free market. This is used for both legitimate (e.g., advertising-financed free services) or questionable purposes (e.g., manipulating voter preferences through personalized messages).

In 2014, The Fappening was an attack that combined phishing and password-guessing to gain access to many user accounts on Apple's iCloud. These acounts included, among other things, backups of all photographs taken with iPhones. Among the private data stolen and published were hundreds of nude pictures of celebrities.

Large Institutions In 2014, Sony Pictures suffered a major break-in (possibly by North Korea to blackmail or punish Sony in relation to the movie *The Interview*) mostly facilitated by unprecedented negligence. Problems included

- unencrypted storage of sensitive information
- password stored in plain text files (sometimes even called "passwords" or placed in the same directory as encrypted files)
- easily guessable passwords
- large number of unmonitored devices
- lack of accountability and responsibility for security, ignorance towards recommendations and audits
- lack of systematic lesson-learning from previous failures (which included 2011 hacks of Sony PlayStation Network and Sony Pictures that stole account information including unsalted or plain text passwords)
- weak IT and information security teams

Stolen data included employee data (including financial data), internal emails, and movies.

In 2016, the US democratic party's headquarters suffered a break-in (possibly by Russia to manipulate or discredit that year's presidential election). The stolen data included in particular internal emails and personal data of donors. Especially, the former hurt the public perception of the party's campaign to an unknown degree that may or may not have been decisive.

User Account Data Many organizations holding user data employ insufficient security against digital break-ins and insufficient (if any) encryption of user data. They get hacked or otherwise compromised so routinely that a strong market for stolen identities has developed, often pricing bulk datasets at a few dollars per identity. Overviews can be found at https://haveibeenpwned.com/ or https://en.wikipedia.org/wiki/SQL_injection.

This development is exacerbated by two human problems:

- System administrators are not sufficiently educated about password hashing and often falsely believe default hash configurations to be secure. Thus, hacks often allow inverting the hash function thus exposing passwords in addition to the possibly sensitive user data.
- Users are not sufficiently educated about systematically using different passwords on every site. Thus, any breach also compromises accounts on any other sites that use the same user name or email address and password.

Many websites now offer and nudge users to use two-factor authentication to protect accounts from identity theft. A second factor (e.g., via email or text message) may be required for every login, for every login from a new location, or for every sensitive action like changing the password. Security questions, which are particularly vulnerable, are phased out by leading companies. But both websites and users are slow to get used to this.

This problem is exacerbated by wide-ranging technically legal access by government agencies to private data. Companies usually comply with subpoenas by the country they operate in (both democracies and others) even if that compromises private user data. For example, in 2017, a US court decided that US companies (i.e., the majority of companies holding sensitive user data) must, if subpoenaed, hand over to US government also all user data stored on servers abroad. That makes it very difficult for other countries to enforce stricter data protection laws.

These constraints interact with market forces and foreign and economic policy. For example, China uses a protectionist policy to strengthen Chinese alternatives to US web services like Google or Facebook. But they also allow foreign companies under the condition that they provide the Chinese government with strong access to private data. Complying with these rules opens Western web service providers up to accusations that they support human rights violations. But dismissing these market opportunities opens them up to competition and shareholder pressure.

The following describes a few high-profile cases of stolen user data:

- In a 2005 hack of the US credit card payment processing company CardSystems, over 40 million accounts were compromised. The stolen data included name and credit card number. The reason was an SQL injection attack.
- In a 2005/2006 hack (reported in 2007) of the US retailer TJX, about 45 million accounts were compromised with an estimated damage of $1 billion. The stolen data included name and credit card number. The cause was the use of the obsolete WEP security standard for communication between pricing devices in one store. This allowed hackers to access the central database, where data was stored unencrypted that should have been deleted.
- In a (estimated) 2008 (only reported in 2016) of myspace, about 360 million accounts were compromised. The stolen data included user name, email address, and badly hashed passwords (unsalted SHA1).
- In a 2012 hack of linkedin, 160 million accounts were compromised. The stolen data included user name, email address, and badly hashed passwords (unsalted SHA1).
- In 2014, data for over 1 billion accounts from $420,000$ websites were discovered by the security company Hold Security. It is claimed that many of these break-ins stems from bot carrying out SQL injection attacks. The details are unclear as some claims by Hold Security are disputed.
- In a 2015 hack of Ashley Madison, about 30 accounts were compromised. The stolen records included name, email address, hashed password, physical description, and sexual preferences. Most passwords were hashed securely (using bcrypt for salting and stretching), but about 10 million passwords were hashed insecurely (using a single MD5 application). This led to multiple extortion attempts and possibly suicides.
- In a 2016 hack of the Friend Finder network, about 400 million accounts were compromised. The stolen records included name, email address, registration date, and unhashed or badly hashed passwords.
- In two separate hacks in 2013 (only reported in 2016) and 2014 of Yahoo, over 1 billion user accounts were compromised by presumably state-sponsored actors. The stolen records included name, email address, phone number, date of birth, and hashed passwords, and in some cases security questions and answers.

3.4.2 Dedicated Systems

Many domains are increasingly using computer technology. Often this is done by engineers with little training in computer science and even less training in security aspects. In many cases, the resulting systems are highly susceptible to attacks, spared only by the priorities of potential hackers and terrorists.

Embedded Systems Embedded systems are increasingly running high-level operating systems, typically variants of Linux or Windows, and software. They are particularly vulnerable due to a number of systemic flaws:

- Software often cannot be updated at all or not conveniently. Thus, they collect many security vulnerabilities over time.
- Affected devices may be in use for years or decades, thus accumulating many vulnerabities.
- It is hard or impossible for users to interact with the software in a way that would allow them to understand or patch its vulnerabilities.
- Access is often not secured or not secured well. Often master passwords (possibly the same on every instance of the system and possibly hard-coded) are used to allow access for technicians.

Cars The upcoming wave of self-driving cars requires the heavy use of experienced software developers and a thorough regulation process. It is therefore reasonable to hope that security will play a major role in the design and legal regulation.

But even today's traditional cars are susceptible to attacks including remote takeover of locks, wheels, or engine. The causes are

- not or not properly protected physical interfaces for diagnostics and repair,
- permanent internet connections, which are useful for navigation and entertainment, that are not strictly separated from engine controls.

One of the more high-profile benevolent attack demonstrations was described in https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

Medical Systems Hospitals and manufacturers of medical devices are notoriously easy to hack.

Weaknesses include unchangeable master passwords, unencrypted communication between devices, outdated and non-updateable software running in devices, and outdated or non-existent protection against attackers. Systemic causes include a highly-regulated release process that precludes fast patching of software and a slow update cycle.

Details: http://cacm.acm.org/magazines/2015/4/184691-security-challenges-for-medical-devices/fulltext

See also the Symantec 2016 Healthcare Internet Security Threat Report available at https://www.symantec.com/solutions/healthcare

Part II

Systematic Software Development

Chapter 4

Implementation

4.1 Process Aspects

This section collects general methods that have been developed by practitioners to get a better handle on managing the software engineering process.
They are generally very cheap and easy to deploy. Therefore, it should[1] be considered gross[2] negligence to develop dependency-critical software without any one of these.
However, training lacks behind with many current developers not updated on latest technologies.

4.1.1 Coding Style

There are many aspects of a program that have no semantic relevance. These include

- most whitespace including
  - indentation and vertical alignment
  - tabs vs. spaces
  - placement of opening and closing brackets
  - optional spaces between operators and arguments
- choice of names for any names that are not fixed in the specification of the interface
  - private methods and field of a class
  - local variables of a function
  - classes and similar units that are not part of the specification
- syntactic restrictions on names including
  - length of names (documenting effect vs. readability)
  - capitalization of first character (e.g., lower case for values, upper case for types)
  - capitalization of inner characters (e.g., camel case vs. underscores)
- syntactic restrictions on declarations including
  - length of a method
  - number of declarations in a class
  - order of declarations (e.g., public vs. private, values vs. methods, mutable vs. immutable)
- formatting of structured documentation including
  - placement of documentation inside the comment
  - use of markdown syntax inside comments
  - presence and order of keywords (e.g., author, param, return)
- placement and formatting of comments containing verification-relevant information including
  - pre/postconditions of methods
  - class invariants
  - loop invariants
  - termination orderings for loops and recursive functions

[1] It is not, though.
[2] *Gross* negligence is the kind that makes people liable to prosecution or litigation.

By standardizing these across a large project, readability is greatly enhanced. This is particularly important when it happens frequently that

- new programmers join a team and have to be quickly retrained on the entire code base
- programmers move between teams
- different teams work on the same code

Style checkers can be standalone or integrated into an IDE. Either way, they allow customizing a style and enforcing it throughout a project.

Modern IDEs (e.g., IntelliJ) provide a wide variety of coding style configurations whose violation results in special warning.

4.1.2 Documentation

Thorough documentation is used for multiple purposes:

- tie the implementation to the specification (e.g., by referencing the exact page or item of the specification corresponding to a declaration)
- inform other programmers about functionality and important subtleties
- provide examples for how to instantiate a class or call a function
- automatically extract web pages containing API documentation
- attach verification-relevant information that can be automatically extracted by verifiers

Most programming languages come with supporting tools that allow for structured documentation. For example, Javadoc is a structured documentation language for Java. The following example snippet is taken from the Javadoc home page:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

Note how it

- uses /** instead of the usual /* to indicate a structured comment
- can link to other code parts (which requires some compilation knowledge to resolve relative references in the documentation)
- integrates HTML which is carried through to the generated web pages
- uses keywords like @param so that better web pages can be produced
- is connected to the code by repeating the names of the function variables (which can be flagged by the style checker)

4.1.3 Versioning

Sophisticated version management systems like svn and recently git have tremendously improved the development process. Specifically, the use of git for the Linux kernel and the success of github have made a huge impact in the open source community.

They allow

- collaboration across physical distances
- maintaining different version of a software (e.g., a release and a development branch)
- using commit messages to log and communicate the reason and effect of a change
- retroactively determining when and by whom a bug was introduced
- automated building and testing on every commit

The following article about google repository management is particularly interesting: http://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

4.1.4 Code Review

Code review is the process of programmers reviewing each other's code before (or sometimes after) it becomes part of the stable parts of the code base.

Many modern versioning tools simplify the process by

- reifying changes (e.g., as diffs/patches or commits) so that each change can be reviewed and applied individually
- managing the available changes and applying them to branches
- maintain the proposed changes and the feedbacks and decisions by the reviewer

Github's maintenance of git pull requests is a simple example of a systematic code review process.

4.1.5 Automated Building and Testing

Most commit-based software repositories allow for hooks that are executed automatically before or after every commit (called push in git).

This is typically used for testing. A typical commit hook should

- create a fresh checkout of the source code
- build the source
- run a test suite (where the test may be part of the source or provided externally)

For most repository managers, automated build managers exist. They can generate reports for every commit and, e.g., alert users by emails upon new commits that fail the tests. A pre-commit hook can even reject the commit if the test failed.

travis is a typical example. It is well-integrated with, e.g., github.

4.1.6 Issue-Tracking

Issue track is the systematic management of known problems, their discussion, and eventual solution. Usually an issue is maintained as a discussion thread, and all open issues are available in a list that allows for filtering and sorting. They are typically provided via web interfaces, in particular to allow users to easily submit issues.

The typical process is

1. The initial post **opens** the issue.
2. After some discussion, the issue is **assigned** to a programmer or team.
3. After posting and checking the solution, the issue is **closed**.

There is a wide variety of issue tracking systems, usually independent of the programming language. Many are integrated with wikis or versioned repositories. Examples are trac and github.

4.2 Programming Aspects

This section collects mostly independent practices for individual aspects of programming.

4.2.1 Input Validation and Internal Syntax

The following is a fundamental principle that is absolutely necessary when handling user input:

- There is a data structure that represents the input/external syntax. This data structure is called the internal syntax. It should exactly follow the grammar of the input language.[3]
- All processing proceeds in the following steps:
  1. User input is parsed from a string holding external syntax into an object of type of the internal syntax. The parser must be side-effect-free: It does not nothing but parse and return an object of the internal syntax or an error message. Failure results in immediately rejecting the user input.
  2. All processing that ever happens works with the internal syntax. External syntax is never visible to any other function than the parser.
  3. The data structure provides a printer (also called serializer) that turns it into a string. Any output that is to be displayed to the user is generated in this way.

It is desirable that parser and printer are exactly inverse to each other. However, it is common that certain aspects of the internal syntax are lost after parsing and printing (e.g., whitespace). However, no meaningful information should be lost, e.g., the parser should not insert default value for omitted optional arguments—instead, it must record that the argument was omitted.

Conversely, parsing followed by printing must succeed and must result in the original object. Thus, we must have $parse(print(i)) = i$ and ideally also $print(parse(e)) = e$.

4.2.2 Common Bugs

We know that correctness is undecidable. But that is irrelevant for a pragmatic approach: the more can be avoided, the better.

Due to undecidability, detecting bugs requires insight and careful case-by-case analyis, which is expensive. Therefore, it has become very successful to focus on one kind of bug and then to systematically find its occurrences. This will usually guarantee bug-freeness but can be very to guard against common bugs.

We can think of these as individual heuristics, each hunting for a specific common bug. Often these heuristics are integrated with the compiler and/or the IDE and can be used to report warnings.

Array and List Bounds When iterating over an array or a list, we often use for-loops with an integer variable that runs from 0 to $n-1$, where n is the length.

Often the length is statically (i.e., at compile-time, without executing) known. For example, if an array is created via $x = Array[int](n)$, we expect for-loop that read or write to x to run from 0 to $n-1$.

If the bounds are different, that can be used to generate a warning.

This is only a heuristic, of course. For the general case, we have to prove that the index i in $x[i]$ is between 0 and $n-1$. There are tools that indeed try to prove that.

An even better solution is to avoid loops altogether that only count up index variables. Instead, we can use *map* or *foreach* operations that traverse an array.

For example, to shift all values in x one up, we can use a counter with subtle traps

```
for i from 1 to x.length − 1
    x[i − 1] := x[i]
x[x.length − 1] := 0
```

A better solution is to use language constructs that enforce the correctness: traversal operators on traversable data structures:

```
x.indices.tail foreach i =>
    x[i − 1] := x[i]
x[−1] := 0
```

[3] Incidentally, this means that untyped languages are out

Here our knowledge about the methods *indices* and *tail* guarantee that the assingment is only executed for indices of elements that have an element before them—no fiddling with the bounds of the for-loop is needed. Moreover, the last assignment uses modular arithmetic to access the last element without fiddling with the length.

Buffer Bounds Buffers can be treated as special arrays. The same techniques apply.

A particularly insane (and useless) property of buffers in certain low level programming languages is that the length of the buffer can be fixed in memory but not fixed in the programming language. Thus, after creating a buffer of size n, we can overread from it, i.e., read $m > n$ values from it. In this case, a C-like language will happily read whatever resides in memory after the buffer.

That can easily be avoided by

- using high-level data structures that hide the memory allocation from users of the data structure,
- or (even better) using high-level programming languages that hide the memory allocation from the programmer entirely.

In the latter case, buffer overreads are at least run-time exceptions.

Null Pointers Null pointers are routinely used by bad programmers, especially in bad programming languages. Analysis tools can inspect every dereferencing of a pointer x and check if there is a previous assignment that assigns a non-null value to x.

A better solution is not to use *null* in the first place. Indeed, good programmers program as if *null* does not exist.

A better solution is to use language constructs that enforce the correctness: the option type

```
data Option[A] = Some(value : A) | None
```

which provides an explicit value for absent values. Now every access to $x : Option[A]$ has to say what to for each of the two possible cases.

The only exception is when calling an external library that uses *null*. But even in that case, it is best to use back-and-forth translations between possibly-null and option values:

```
fun fromMaybeNull[A](x : A) : Option[A] =
    if x == null
        None
    else
        Some(x)
fun toMaybeNull[A](x : Option[A]) : A =
    x.getOrElse(null)
```

Casting Analysis tools can inspect every type cast x **asInstanceOf** B of a value x to type B. They can infer the type of x, say A, and check for

- plausibility: is B a subtype of A—if not, the cast is definitely a bug,
- correctness: is x guaranteed to have type B—that requires a proof.

Such casts are typically guarded as in

```
if x isInstanceOf B
    f(B asInstanceOf )
else
    g(x)
```

A better solution is to use language constructs that enforce the correctness. One way to do this is a case-distinction operator

```
match x
    x : B ↦ f(x)
    _ ↦ g(x)
```

which allows the compiler to spot a missing case if the second case is forgotten.

An even simpler solution is to use a cast operator that returns an options value:

```
fun optCast(x : A) : Option[B] =
    ...
(optCast(x) map f).getOrElse(g(x))
```

Uninitialized Memory Some programming languages allow introducing names without initial values. Those can be variables or (in C-like low-level languages) memory areas.

Uninitialized variables can easily be spotted by analysis tools. They should not be warnings but actual compiler bugs. Allowing them at all is a design flaw of the programming language.

A variant uninitialized variables are variables initialized with *null*. That is equally easy to spot and equally forbidden.

Uninitialized memory areas are occasionally useful when initializing a large memory area is considered too costly. In most cases, this can be avoided entirely by using good data structures. For example, there is no need to use that is known to be filled later anyway Those may be variables, which are simply declared without value. That

Unreachable Code If no execution can ever execute a certain command, we speack of unreachable code. It is always a bug.

Many instances of unreachable code can be detected automatically. This includes

- code in a branch of an if-else whose condition always has the same value
- code in a case distinction (switch) statement that come after a default case
- code after a return statnent

4.2.3 Safe by Design

This section collects a few implementation principles that help minimize the likely hood of errors.

Safe Defaults Default and initial values should always be chosen in such a way that they lead to the minimal possible behavior.

For example, a security check should be wrapped in an exception handler that treats every exception as failure of the check.

Minimal Interfaces and Access Rights Any component C that needs access to critical shared component D should have only the minimal access rights needed for its correct operation.

For example, a separate abstract class D' can be written that contains only those methods of D that C needs. C should then by implemented against D' rather than D.

If D is a database, file system, or similar external resource, we should write special functions that wrap around the access to D in exactly the way needed by C. For example, if C often has to append to a file, we write a function for appending to a file; C will call only that function to access the file, and no other part of C makes any file system access.

4.3 Stability

Frequent changes are extremely dangerous to even the best software development process.

Stability is particularly important for

- the specification,
- the design of the data structures and algorithms,
- the project team,
- the coding style,
- the workflows for building, committing, etc.,
- the policies for access, review, and approval of changes.

It is very tempting for managers, marketing department, and customers to request changes because they seem easy to them. Even many programmers often underestimate their dangers.

But every change at a high level introduces lots of increasingly greater and more expensive changes at lower levels. Eventually, the lower levels (especially when resources are tight, which is always the case) have to introduce workarounds, hacks, and special-case treatments to handle the changes.

To the top-level person who requested the change, everything looks fine because the lower levels will usually do a good job of hiding the mess. But below the surface the codebase will become increasingly messy until it is unmanagable.

A good metaphor is to think of a long stick representing the hiearchy. The person at the top points the stick at a slightly different angle, which does not very different from the top. But at the lower end of the stick, the small change in angle caused a massive shift of the end of the stick. The shift may be much bigger than what the inertia of the stick allows, and the sticks breaks somewhere in the middle. When the stick breaks, the people at the breaking point in the middle move the lower half of the stick so that it points to the right point and hire two new people A and B. A constantly measures the movement of the upper half of the stick and shouts the values to B. B then computes how much the lower half would move if the halves were still connected and moves the lower half accordingly. The person at the top does not notice anything. But over time, the stick has broken into many pieces, and lots of people are in charge of pretending it is still one piece.

Chapter 5

Dynamic Analysis (Testing)

Chapter 6

Static Analysis

Part III

Specification-Near Programming

Chapter 7

Type Theory

Chapter 8

Functional Programming

Chapter 9

Combining Logic and Programming

Part IV

Formal Methods

Chapter 10

Program Synthesis

Chapter 11

Model Checking

Chapter 12

Theorem Proving

Part V

Security

Chapter 13

Social Engineering

Chapter 14

Common Criteria

Chapter 15

Cryptography

15.1 History

15.2 Theory of symmetric encryption

15.2.1 Preliminaries

Definition 15.1 (polynomial-time algorithm)**.** An algorithm A is called polynomial time algorithm iff there exists $k \in \mathbb{N}$ such that the maximum number t of operations A has to perform for input of length n satisfies $t(n) \in O(n^k)$.

Definition 15.2 (A probabilistic polynomial-time algorithm)**.** A probabilistic polynomial-time algorithm (PPT) is a polynomial-time algorithm that might be non-deterministic. If it is deterministic we will just call it polynomial time algorithm.

Definition 15.3. A function $f : \mathbb{N} \to \mathbb{R}$ is called negligible iff

$$\forall k \in \mathbb{N}.\, \exists N_k \in \mathbb{N}.\, \forall n \geq N_k :\, n\,|f(n)| < 1.$$

Definition 15.4. An encryption scheme is an ordered triple (G, E, D), where G and E are PPt algorithms and D is a polynomial time algorithm iff for any *security parameter* $n \in \mathbb{N}$:

- The *key generation algorithm* G, takes as input 1 and uses randomness to chose a *key* k from a set of possible keys K_n (the key space)
- The *encryption algorithm* E, takes as input a *message* $m \in P_n$ (P_n is called the *plaintext space* for the security parameter n) and uses k to compute an encrypted message $c \in C_n$ (C_n is called the *ciphertext space* for n). If E uses the key k on the message m and outputs s, we write $E_k(m) = c$.
- The *decryption algorithm* D, takes an encrypted message and uses the key k to decipher it (deterministically). So we have
$$\forall n \in \mathbb{N},\, \forall m \in P_n,\, \forall k \in K_n.\, D_k(E_k(m)) = m.$$

15.2.2 Notions of secure Encryption

There are various different notions of "security" of encryption. We will discuss the notions of computational indistinguishability (comp. ind.) and security against a chosen Plaintext attack (ind. CPA).

Definition 15.5 (computational indistinguishable)**.** We will call an encryption scheme (E, D, k) guess indistinguishable iff for any PPT A, for any two messages m_1, m_2 of length n and $i \in \{0,1\}$, uniformly random chosen, there is a negligible function *neg.* s.t.:

$$(\Pr[A(E(m_0)) = 1] - \Pr[A(E(m_1)) = 1]) \leq \frac{1}{2} + neg(n),$$

where $neg(n)$ is a negligible function.

15.2.3 Realization of secure encryption

One important cryptographic primitive is the so called one-way function (OWF). Using these one-way-functions one can build provably secure symmetric encryption algorithms. However, it is currently not known whether there exist any one-way function. In the following we will especially consider one-way permutations (OWP), OWFs with identical input and output length.

Definition 15.6 (One way function)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$, is called a one-way function iff, f is a PPT and for any natural number n and any PPT A, there is a negligible function *neg* such that:

$$\Pr[f(A(f(x), 1^n)) = f(x)] \leq neg,$$

where x is chosen uniformly random.

Intuitively, OWFs are simply hard to invert PPTs. Some functions that are commonly believed to be OWFs are modular exponentiation and the multiplication of big prime numbers. Another important concept is the so called Pseudo-random generator (PRG or PRNG).

Definition 15.7 (PRGs)**.** A function $H : \{0,1\}^* \to \{0,1\}^*$ is called PRG iff:

- H is a PPT
- There exists a PPT the so called *length extension function* $l : \mathbb{N} \to \mathbb{N}$, s.t. $\forall n \in \mathbb{N}.\, l(n) > n \wedge |H(x)| = l(|x|), \forall x \in \{0,1\}^*$
- There is a negligible function *neg* s.t. for any PPT A, we have:

  $$\left|\Pr[A(H(U_n)) = 1] - \Pr\left[D(U_{l(n)}) = 1\right]\right| < neg(n),$$

  where U_k denotes a uniformly random element of $\{0,1\}^k$.

It can be shown that given any OWF, we can build a PRG. Given a PRG, we can iterate it on its own output to get an arbitrarily long pseudo-random output. Now we can construct an encryption scheme by simply taking the key as input to a pseudo random generator and xoring the message with the output of the pseudo-random generator. The decryption is the can be done by simply encrypting the ciphertext a second time. The resulting encryption scheme can already be shown to be computational indistinguishable, but not necessarily ind. CPA secure. We will now try to improve the security to also reach at least ind. CPA security. For this we can use a so called Feistel network.

Definition 15.8 (A Feistel cipher)**.** Let k be any odd natural number (the number of *rounds*). Let f_{k_i} be a family of functions (f is the so called *round function*) of output length n indexed by the sequence of *round keys* $k_1, k_2, \ldots, k_n$. Then the following encryption algorithm E_k is called Feistel cipher.

- Fix a message $m =: x_1 \circ x_2$, where $|x_1| = |x_2| = n$
- Define the sequences $L_1, L_2, \ldots, L_n$ and $R_1, R_2, \ldots, R_n$ by $L_1 := x_1, R_1 := x_2$ and $L_{n+1} := R_n, R_{n+1} := L_n \oplus f_{k_n}(R_n)$. Finally define $E_k : x_1 \circ x_2 \to L_k \circ R_k$.

Now we can define the corresponding decryption algorithm D_k just like e_k, but with the reversed order of round keys:

- Fix a ciphertext $c =: x_1 \circ x_2$, where $|x_1| = |x_2| = n$
- Define the sequences $L_1, L_2, \ldots, L_n$ and $R_1, R_2, \ldots, R_n$ by $L_1 := x_1, R_1 := x_2$ and $L_{n+1} := R_n, R_{n+1} := L_n \oplus f_{k_{k-n}}(R_n)$. Finally define $D_k : x_1 \circ x_2 \to L_k \circ R_k$.

Feistel ciphers have been shown to fulfill several notions of security assuming that the round function is actually pseudo random. For instance Feistel networks with at least 3 rounds are ind. CPA secure and for more rounds they fulfill even stronger notions of security.

15.2.4 Realization in actually used encryption

In practice many symmetric encryption schemes are based either on Feistel networks or on substitution-permutation-networks.

Definition 15.9 (Substitution-permutation-network)**.** A substitution-permutation-networks is an series of linked substitutions (*S-Boxes*) and permutations *P-Boxes* of blocks in an encryption algorithm. The corresponding encryption algorithm splits the message into several boxes with are typically fed into the S-Boxes then the result is fed into the P-Box (and at some point the round key is used, for instance xored with the result as is AES). The later mentioned AES-encryption algorithm is one example of an encryption algorithm build around a substitution-permutation-network.

Substitution-permutation-network and Feistel networks using S-Boxes are quite similar, but there are also a few differences. Ciphers based on substitution-permutation-network can be better parallelized, but Feistel ciphers can use any pseudo random function (for instance any one way function) and is not limited to invertible (P-Boxes).

15.3 Symmetric Encryption

15.3.1 AES

15.4 Asymmetric Encryption

The idea behind RSA is that if $N = p \cdot q$ for large prime numbers p and q, it is very difficult to compute p and q from n.

15.4.1 RSA

Setup Choose two large primes p and q (typically of roughly equal size). Put $N = p \cdot q$.

Now put $n = (p-1)(q-1)$. (Actually, any common multiple of the two numbers is fine.) Note that $n = \varphi(N)$. Pick $e \in Z_n$ such that there is a $d \in \mathbb{Z}_n$ with $e \cdot d \equiv_n 1$. Such a d exists if $\gcd(e, n) = 1$ and is easy to compute (see Thm. A.14).

The keys are defined as follows:

- public information (encryption key): N and e
- private information (decryption key): n, d, p, and q

Among the private information, only N and d are needed later on. So n, p, and q can be forgotten. But they have to remain private—p (or q) is enough to compute n and d.

Different keys are often compared by their size. That size is the number of bits in N.

Encryption Messages are numbers $x \in F_N$. For example, we can choose the largest k such that $2^k < N$ and use k-bit messages.

Encryption and decryption are functions $\mathbb{Z}_N \to \mathbb{Z}_N$ given by

- encryption: $x \mapsto x^e \bmod N$
- decryption: $x \mapsto x^d \bmod N$

These are indeed inverse to each other:

Theorem 15.10. *For all $x \in Z_N$, we have $(x^d)^e \equiv_N (x^e)^d \equiv_N x$.*

Proof. In general, because $N = p \cdot q$ for prime numbers p and q, we have that $x \equiv_N y$ iff $x \equiv_p y$ and $x \equiv_q y$.
So we have to show that $x^{de} \equiv_p x$. (We also have to show the same result for q, but the proof is the same.) We distinguish two cases:

- $p|x$: Then trivially $x^{de} \equiv_p x \equiv_p 0$.
- Otherwise. Then p and x are coprime.
  By construction of e and d and using Thm. A.14, we have $k \in \mathbb{N}$ such that $e \cdot d + k \cdot n = 1$. Thus, we have to show $x^{de} = x \cdot (x^{p-1})^{k \cdot (q-1)} \equiv_p x$. That follows from $x^{p-1} \equiv_p 1$ as known from Thm. A.25.

□

Attacks To break RSA, d has to be computed. There are 3 natural ways to do that:

- Factor N into p and q. Then compute d easily.
- Compute n using $n = \varphi(N)$ (which may be easier than finding p and q). Then compute d easily.
- Find d such that $e \cdot d \equiv_n 1$ (which may be easier than finding n).

Currently these are believed to be equally hard.

It is believed that there is no algorithm for factoring N that is polynomial in the number of bits of N. That is no proved. There are hypothetical machines (e.g., quantum computers) that can factor N polynomially.

Note that checking if N can be factored (without producing the factors) is polynomial, and practical algorithms exist (in particular, the AKS algorithms). That is important to find the large prime number p and q efficiently.

If there is indeed no polynomial algorithm, factoring relies on brute-force attacks that find all prime numbers $k < \sqrt{N}$ and test $k|N$. Therefore, larger keys are harder than break to smaller ones. Because of improving hardware, the key size that is considered secure grows over time.

Keys of size 1024 are considered secure today, but because security is a relative term, keys of size 2048 are often recommended. Larger keys are especially important if data is needed to remain secure far into the future, when faster hardware will be available.

15.5 Authentication

15.6 Hashing

15.6.1 MDx

15.6.2 SHA-x

15.7 Key Generation and Distribution

Chapter 16

Privacy

Part VI

Appendix

Appendix A

Mathematical Preliminaries

A.1 Binary Relations

A binary relation is on A is a subset $\# \subseteq A \times A$. We usually write $(x, y) \in \#$ as $x\#y$.

A.1.1 Classification

Definition A.1 (Properties of Binary Relations)**.** We say that $\#$ is ... if the following holds:

- reflexive: for all x, $x\#x$
- irreflexive: for no x, $x\#x$
- transitive: for all x, y, z, if $x\#y$ and $y\#z$, then $x\#z$
- a strict order: irreflexive and transitive
- a preorder: reflexive and transitive
- anti-symmetric: for all x, y, if $x\#y$ and $y\#x$, then $x = y$
- symmetric: for all x, y, if $x\#y$, then $y\#x$
- an order[1]: preorder and anti-symmetric
- an equivalence: preorder and symmetric
- a total order: order and for all x, y, $x\#y$ or $y\#x$

An element $a \in A$ is called ... of $\#$ if the following holds:

- least element: for all x, $a\#x$
- greatest element: for all x, $x\#a$
- least upper bound for x, y: $x\#a$ and $y\#a$ and for all z, if $x\#z$ and $y\#z$, then $a\#z$
- greatest lower bound for x, y: $a\#x$ and $a\#y$ and for all z, if $z\#x$ and $z\#y$, then $z\#a$

Definition A.2 (Dual Relation)**.** For every relation $\#$, the relation $\#^{-1}$ is defined by $x \#^{-1} y$ iff $y \# x$.
$\#^{-1}$ is called the **dual** of $\#$.

Theorem A.3 (Dual Relation)**.** *If a relation is reflexive/irreflexive/transitive/symmetric/antisymmetric/total, then so is its dual.*

A.1.2 Equivalence Relations

Equivalence relations are usually written using infix symbols whose shape is reminiscent of horizontal lines, such as $=$, $\sim$, or $\equiv$. Often vertically symmetric symbols are used to emphasize the symmetry property.

Definition A.4 (Quotient)**.** Consider a relation $\equiv$ on A. Then

- For $x \in A$, the set $\{y \in A \mid x \equiv y\}$ is called the (equivalence) **class** of x. It is often written as $[x]_{\equiv}$.
- $A/\equiv$ is the set of all classes. It is called the **quotient** of A by $\equiv$.

Theorem A.5. *For a relation $\equiv$ on A, the following are equivalent[2]:*

- *$\equiv$ is an equivalence.*
- *There is a set B and a function $f : A \to B$ such that $x \equiv y$ iff $f(x) = f(y)$.*
- *Every element of A is in exactly one class in $A/\equiv$.*

In particular, the elements of $A/\equiv$

- *are pairwise disjoint,*
- *have A as their overall union.*

A.1.3 Orders

Theorem A.6 (Strict Order vs. Order)**.** *For every strict order $<$ on A, the relation "$x < y$ or $x = y$" is an order.*
For every order $\leq$ on A, the relation "$x \leq y$ and $x \neq y$" is a strict order.

Thus, strict orders and orders come in pairs that carry the same information.
Strict orders are usually written using infix symbols whose shape is reminiscent of a semi-circle that is open to the right, such as $<$, $\subset$, or $\prec$. This emphasizes the anti-symmetry ($x < y$ is very different from $y < x$.) and the transitivity ($< \ldots <$ is still $<$.) The corresponding order is written with an additional horizontal bar at the bottom, i.e., $\leq$, $\subseteq$, or $\preceq$. In both case, the mirrored symbol is used for the dual relation, i.e., $>$, $\supset$, or $\succ$, and $\geq$, $\supseteq$, and $\succeq$.

Theorem A.7. *If $\leq$ is an order, then least element, greatest element, least upper bound of x, y, and greatest lower bound of x, y are unique whenever they exist.*

Theorem A.8 (Preorder vs. Order)**.** *For every preorder $\leq$ on A, the relation "$x \leq y$ or $y \leq x$" is an equivalence.*
For equivalence classes X and Y of the resulting quotient, $x \leq y$ holds for either all pairs or no pair $(x, y) \in X \times Y$. If it holds for all pairs, we write $X \leq Y$ as well.
The relation $\leq$ on the quotient is an order.

A.2 Binary Functions

A binary function on A is a function $\circ : A \times A \to A$. We usually write $\circ(x, y)$ as $x \circ y$.

Definition A.9 (Properties of Binary Functions)**.** We say that $\circ$ is ... if the following holds:

- associative: for all x, y, z, $x \circ (y \circ z) = (x \circ y) \circ z$
- commutative: for all x, y, $x \circ y = y \circ x$
- idempotent: for all x, $x \circ x = x$

An element $a \in A$ is called a ... element of $\circ$ if the following holds:

- left-neutral: for all x, $a \circ x = x$
- right-neutral: for all x, and $x \circ a = x$
- neutral: left-neutral and right-neutral
- left-absorbing: for all x, $a \circ x = a$
- right-absorbing: for all x, $x \circ a = a$
- absorbing: left-absorbing and right-absorbing

Theorem A.10. *Neutral and absorbing element of $\circ$ are unique whenever they exist.*

A.3 The Integer Numbers

A.3.1 Divisibility

Definition A.11 (Divisibility)**.** For $x, y \in \mathbb{Z}$, we write $x|y$ iff there is a $k \in \mathbb{Z}$ such that $x * k = y$.
We say that y is divisible by x or that x divides y.

Remark A.12 (Divisible by 0 and 1). Even though division by 0 is forbidden, the case $x = 0$ is perfectly fine. But it is boring: $0|x$ iff $x = 0$.
Similarly, the case $x = 1$ is trivial: $1|x$ for all x.

Theorem A.13 (Divisibility)**.** *Divisibility has the following properties for all $x, y, z \in Z$*

- *reflexive:* $x|x$
- *transitive: if* $x|y$ *and* $y|z$ *then* $x|z$
- *anti-symmetric for natural numbers* $x, y \in \mathbb{N}$*: if* $x|y$ *and* $y|x$*, then* $x = y$
- 1 *is a least element:* $1|x$
- 0 *is a greatest element:* $x|0$
- $\gcd(x, y)$ *is a greatest lower bound of* x, y
- $\mathrm{lcm}(x, y)$ *is a least upper bound of* x, y

Thus, $|$ *is a preorder on* $\mathbb{Z}$ *and an order on* $\mathbb{N}$.

Divisibility is preserved by arithmetic operations: If $x|m$ *and* $y|m$*, then*

- *preserved by addition:* $x + y|m$
- *preserved by subtraction:* $x - y|m$
- *preserved by multiplication:* $x * y|m$
- *preserved by division if* $x/y \in Z$*:* $x/y|m$
- *preserved by negation of any argument:* $-x|m$ *and* $x| - m$

gcd *has the following properties for all* $x, y \in \mathbb{N}$*:*

- *associative:* $\gcd(\gcd(x, y), z) = \gcd(x, \gcd(y, z))$
- *commutative:* $\gcd(x, y) = \gcd(y, x)$
- *idempotence:* $\gcd(x, x) = x$
- 0 *is a neutral element:* $\gcd(0, x) = x$
- 1 *is an absorbing element:* $\gcd(1, x) = 1$

lcm *has the same properties as* gcd *except that* 1 *is neutral and* 0 *is absorbing.*

Theorem A.14. *For all $x, y \in \mathbb{Z}$, there are numbers $a, b \in \mathbb{Z}$ such that $ax + by = \gcd(x, y)$.*
a and b can be computed using the extended Euclidean algorithms.

Definition A.15. If $\gcd(x, y) = 1$, we call x and y **coprime**.
For $x \in \mathbb{N}$, the number of coprime $y \in \{0, \ldots, x - 1\}$ is called $\varphi(x)$. φ is called Euler's **totient function**.

We have $\varphi(0) = 0$, $\varphi(1) = \varphi(2) = 1$, $\varphi(3) = 2$, $\varphi(4) = 1$, and so on. Because $\gcd(x, 0) = x$, we have $\varphi(x) \leq x - 1$. x is prime iff $\varphi(x) = x - 1$.

A.3.2 Equivalence Modulo

Definition A.16 (Equivalence Modulo)**.** For $x, y, m \in \mathbb{Z}$, we write $x \equiv_m y$ iff $m|x - y$.

Theorem A.17 (Relationship between Divisibility and Modulo)**.** *The following are equivalent:*

- $m|n$
- $\equiv_m \supseteq \equiv_n$ *(i.e., for all x, y we have that $x \equiv_n y$ implies $x \equiv_m y$)*
- $n \equiv_m 0$

Remark A.18 (Modulo 0 and 1). In particular, the cases $m = 0$ and $m = 1$ are trivial again:

- $x \equiv_0 y$ iff $x = y$,
- $x \equiv_1 y$ always

Thus, just like 0 and 1 are greatest and least element for $|$, we have that $\equiv_0$ and $\equiv_1$ are the smallest and the largest equivalence relation on $\mathbb{Z}$.

Theorem A.19 (Modulo)**.** *The relation $\equiv_m$ has the following properties*

- *reflexive: $x \equiv_m x$*
- *transitive: if $x \equiv_m y$ and $y \equiv_m z$ then $x \equiv_m z$*
- *symmetric: if $x|y$ then $y|x$*

Thus, it is an equivalence relation.

It is also preserved by arithmetic operations: If $x \equiv_m x'$ and $y \equiv_m y'$, then

- *preserved by addition: $x + y \equiv_m x' + y'$*
- *preserved by subtraction: $x - y \equiv_m x' - y'$*
- *preserved by multiplication: $x * y \equiv_m x' * y'$*
- *preserved by division if $x/y \in Z$ and $x'/y' \in \mathbb{Z}$: $x/y \equiv_m x'/y'$*
- *preserved by negation of both arguments: $-x \equiv_m -x'$*

A.3.3 Arithmetic Modulo

Definition A.20 (Modulus)**.** We write $x \bmod m$ for the smallest $y \in \mathbb{N}$ such that $x \equiv_m y$.
We also write $modulus_m$ for the function $x \mapsto x \bmod m$. We write $\mathbb{Z}_m$ for the image of $modulus_m$.

Remark A.21 (Modulo 0 and 1). The cases $m = 0$ and $m = 1$ are trivial again:

- $x \bmod 0 = x$ and $\mathbb{Z}_0 = \mathbb{Z}$
- $x \bmod 1 = 0$ and $\mathbb{Z}_1 = \{0\}$

Remark A.22 (Possible Values). For $m \neq 0$, we have $x \bmod m \in \{0, \dots, m-1\}$. In particular, there are m possible values $m \bmod x$.
For example, we have $x \bmod 1 \in \{0\}$. And we have $x \bmod 2 = 0$ if x is even and $x \bmod 2 = 1$ if x is odd.

Definition A.23 (Arithmetic Modulo m)**.** For $x, y \in \mathbb{Z}$, we define arithmetic operations modulo m by

$$x \circ_m y = (x \circ y) \bmod m \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, if there is a unique $q \in \mathbb{Z}_m$ such that $q \cdot x \equiv_m y$, we define $x/_m y = q$.

Note that the condition $y|x$ is neither necessary nor sufficient for $x/_m y$ to de defined. For example, $2/_4 2$ is undefined because $1 \cdot 2 \equiv_4 3 \cdot 2 \equiv_4 2$. Conversely, $2/_4 3$ is defined, namely 2.

Theorem A.24 (Arithmetic Modulo m)**.** *For $x, y \in \mathbb{Z}$,* mod *commutes with arithmetic operations in the sense that*

$$(x \circ y) \bmod m = (x \bmod m) \circ_m (y \bmod m) \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, $x/_m y$ is defined iff $\gcd(y, m) = 1$ *and*

$$(x/y) \bmod m = (x \bmod m)/_m (y \bmod m) \qquad \text{if} \qquad y|x$$

$$x/_m y = x \cdot_m a \qquad \text{if } ay + bm = 1 \text{ as in see Thm. A.14}$$

Theorem A.25 (Fermat's Little Theorem)**.** *For all prime numbers p and $x \in \mathbb{Z}$, we have that $x^p \equiv_p x$.*
If x and p are coprime, that is equivalent to $x^{p-1} \equiv 1$.

A.3.4 Digit-Base Representations

Fix $m \in \mathbb{N} \setminus \{0\}$, which we call the base.

Theorem A.26 (Div-Mod Representation)**.** *Every $x \in \mathbb{Z}$ can be uniquely represented as $a \cdot m + b$ for $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_m$.*
Moreover, $b = x \bmod m$. We write $b \operatorname{div} m$ for a.

Definition A.27 (Base-m-Notation)**.** For $d_i \in \mathbb{Z}_m$, we define $(d_k \ \ldots \ d_0)_m = d_k \cdot m^k + \ldots + d_1 \cdot k + d_0$.
The d_i are called digits.

Theorem A.28 (Base-m Representation)**.** *Every $x \in \mathbb{N}$ can be uniquely represented as $(0)_m$ or $(d_k \ \ldots \ d_0)_m$ such that $d_k \neq 0$.*
Moreover, we have $k = \lfloor \log_m x \rfloor$ and $d_0 = x \bmod m$, $d_1 = (x \operatorname{div} m) \bmod m$, $d_2 = ((x \operatorname{div} m) \operatorname{div} m) \bmod m$ and so on.

Example A.29 (Important Bases). We call $(d_k \ \ldots \ d_0)_m$ the binary/octal/decimal/hexadecimal representation if $m = 2, 8, 10, 16$, respectively.
In case $m = 16$, we write the elements of $\mathbb{Z}_m$ as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

A.3.5 Finite Fields

In this section, let $m = p$ be prime.

Construction Then $x/_p y$ is defined for all $x, y \in \mathbb{Z}_p$ with $y \neq 0$. Consequently, $\mathbb{Z}_p$ is a field.
Up to isomorphism, all finite fields are obtained as an n-dimensional vector space $\mathbb{Z}_p^n$ for $n \geq 1$. This field is usually called F_{p^n} because it has p^n elements. From now on, let $q = p^n$.
All elements of F_q are vectors $(a_0, \ldots, a_{n-1})$ for $a_i \in \mathbb{Z}_p$. Addition and subtraction are component-wise, the 0-element is $(0, \ldots, 0)$, the 1-elements is $(1, 0, \ldots, 0)$.
However, multiplication in F_q is tricky. To multiply two elements, we think of the vectors $(a_0, \ldots, a_{n-1})$ as polynomials $a_{n-1}X^{n-1} + \ldots + a_1 X + a_0$, and multiply the polynomials. This can introduce powers X^n and higher, which we eliminate using $X^n = k_{n-1}X^{n-1} + \ldots + k_1 X + k_0$. The resulting polynomial has degree at most $n - 1$, and its coefficient (modulo p) yield the result.
The values k_i always exists but are non-trivial to find. They must be such that the polynomial $X^n - k_{n-1}X^{n-1} - \ldots - k_1 X - k_0$ has no roots in $\mathbb{Z}_p$. There may be multiple polynomials, which may lead to different multiplication operations. However, all of them yield isomorphic fields.

Binary Fields The operations become particularly easy if $p = 2$. The elements of F_{2^n} are just the bit strings of length n. Addition and subtraction are the same operation and can be computed by component-wise XOR. Multiplication is a bit more complex but can be obtained as a sequence of bit-shifts and XORs.

Exponentiation and Logarithm Because F_q has multiplication, we can define natural powers in the usual way:

Definition A.30. For $x \in F^q$ and $l \in \mathbb{N}$, we define $x^l \in F_q$ by $x^0 = 1$ and $x^{l+1} = x \cdot x^l$.
If l is the smallest number such that $x^l = y$, we write $l = \log_x y$ and call n the **discrete q-logarithm** of y with base x.

The powers $1, x, x^2, \ldots \in F_q$ of x can take only $q-1$ different values because F_q has only q elements and x^l can never be 0 (unless $x = 0$). Therefore, they must be periodic:

Theorem A.31. *For every $x \in F_q$, we have $x^q = x$ or equivalently $x^{q-1} = 1$ for $x \neq 0$.*

For some x, the period is indeed $q-1$, i.e., we have $\{1, x, x^2, \ldots, x^{q-1} = F_q \setminus \{0\}$. Those x are called primitive elements of F_q. But the period may be smaller. For example, the powers of 1 are $1, \ldots, 1$, i.e., 1 has period 1. For a non-trivial example consider $p = 5$, $n = 1$, (i.e., $q = 5$): The powers of 4 are $4^0 = 1$, $4^1 = 4$, $4^2 = 16 \bmod 5 = 1$, and $4^3 = 4$.
If the period is smaller, x^l does not take all possible values in F_q. Therefore, $\log_x y$ is not defined for all $y \in F_q$.

Computing x^l is straightforward and can be done efficiently. (If $n > 1$, we first have to find the values k_i needed to do the multiplication, but we can precompute them once and for all.)
Determining whether $\log_x y$ is defined and computing its value is also straightforward: We can enumerate all powers $1, x, x^2, \ldots$ until we find 1 or y. However, no efficient algorithm is known.

A.4 Size of Sets

The size $|S|$ of a set S is a very complex topic of mathematics because there are different degrees of infinity. Specifically, we have that $|\mathcal{P}(S)| > |S|$, i.e., we have infinitely many degrees of infinity.
In computer science, we are only interested in countable sets. We use a very simple definition that writes C for countable and merges all greater sizes into uncountable sets, whose size we write as U.

Definition A.32 (Size of sets)**.** The size $|S| \in \mathbb{N} \cup \{C, U\}$ of a set S is defined by:

- if S is finite: $|S|$ is the number of elements of S
- if S is infinite and bijective to $\mathbb{N}$: $|S| = C$, and we say that S is countable
- if S is infinite and not bijective to $\mathbb{N}$: $|S| = U$, and we say that S is uncountable

We can compute with set sizes as follows:

Definition A.33 (Computing with Sizes)**.** For two sizes $s, t \in \mathbb{N} \cup \{C, U\}$, we define addition, multiplication, and exponentiation by the following tables:

$s+t$	t: $n \in \mathbb{N}$	C	U
$m \in \mathbb{N}$	$m+n$	C	U
C	C	C	U
U	U	U	U

$s*t$	t: $n \in \mathbb{N}$	C	U
$m \in \mathbb{N}$	$m*n$	C	U
C	C	C	U
U	U	U	U

s^t	t: 0	1	$n \in \mathbb{N} \setminus \{0\}$	C	U
0	1	0	0	0	0
1	1	1	1	1	1
$m \in \mathbb{N} \setminus \{0\}$	1	m	m^n	U	U
C	1	C	C	U	U
U	1	U	U	U	U

Because exponentiation s^t is not commutative, the order matters: s is given by the row and t by the column.

The intuition behind these rules is given by the following:

Theorem A.34. *For all sets S, T, we have for the size of the*

- *disjoint union:*
$$|S \uplus T| = |S| + |T|$$
- *Cartesian product:*
$$|S \times T| = |S| * |T|$$
- *set of functions from T to S:*
$$|S^T| = |S|^{|T|}$$

Thus, we can understand the rules for exponentiation as follows. Let us first consider the 4 cases where one of the arguments has size 0 or 1: For every set A

1. there is exactly one function from the empty set (namely the empty function): $|A^{\varnothing}| = 1$,
2. there are as many functions from a singleton set as there are elements of A: $|A^{\{x\}}| = |A|$,
3. there are no functions to the empty set (unless A is empty): $|\varnothing^A| = 0$ if $A \neq \varnothing$,
4. there is exactly one function into a singleton set (namely the constant function): $|\{x\}^A| = 1$,

Now we need only one more rule: The set of functions from a non-empty finite set to a finite/countable/uncountable set is again finite/countable/uncountable. In all other cases, the set of functions is uncountable.

A.5 Important Sets and Functions

The meaning and purpose of a data structure is to describe a set in the sense of mathematics. Similarly, the meaning and purpose of an algorithm is to describe a function between two sets.

Thus, it is helpful to collect some sets and functions as examples. These are typically among the first data structures and algorithms implemented in any programming language and they serve as test cases for evaluating our languages.

A.5.1 Base Sets

When building sets, we have to start somewhere with some sets that are assumed to exist. These are called the *bases sets* or the *primitive sets*.

The following table gives an overview, where we also list the size of each set according to Def. A.32:

set	description/definition	size
typical base sets of mathematics[3]		
$\varnothing$	empty set	0
$\mathbb{N}$	natural numbers	C
$\mathbb{Z}$	integers	C
$\mathbb{Z}_m$ for $m > 0$	integers modulo m, $\{0, \ldots, m-1\}$ [4]	m
$\mathbb{Q}$	rational numbers	C
$\mathbb{R}$	real numbers	U
additional or alternative base sets used in computer science		
unit	unit type, $\{()\}$, equivalent to $\mathbb{Z}_1$	1
$\mathbb{B}$	booleans, $\{false, true\}$, equivalent to $\mathbb{Z}_2$	2
int	primitive integers, $-2^{n-1}, \ldots, 2^{n-1} - 1$ for machine-dependent n, equivalent to $\mathbb{Z}_{2^n}$ [5]	2^n
float	IEEE floating point approximations of real numbers	C
char	characters	finite[6]
string	lists of characters	C

[3] All of mathematics can be built by using $\varnothing$ as the only base set because the others are definable. But it is common to assume at least the number sets as primitives.

[4] $\mathbb{Z}_0$ also exists but is trivial: $\mathbb{Z}_0 = \mathbb{Z}$.

[5] Primitive integers are the 2^n possible values for a sequence of n bits. Old machines used $n = 8$ (and the integers were called "bytes"), later machines used $n = 16$ (called "words"). Modern machines typically use 32-bit or 64-bit integers. Modern programmers usually—but dangerously—assume that 2^n is much bigger than any number that comes up in practice so that essentially $int = \mathbb{Z}$.

[6] The ASCII standard defined 2^7 or 2^8 characters. Nowadays, we use Unicode characters, which is a constantly growing set containing

A.5.2 Functions on the Base Sets

For every base set, we can define some basic operations. These are usually built-in features of programming languages whenever the respective base set is built-in.
We only list a few examples here.

Numbers

For all number sets, we can define addition, subtraction, multiplication, and division in the usual way.
Some care must be taken when subtracting or dividing because the result may be in a different set. For example, the difference of two natural numbers is not in general a natural number but only an integer (e.g., $3 - 5 \notin \mathbb{N}$). Moreover, division by 0 is always forbidden.

Quotients of the Integers

The function $modulus_m$ (see Sect. A.3.3) for $m \in \mathbb{N}$ maps $x \in \mathbb{Z}$ to $x \bmod m \in \mathbb{Z}_m$.
In programming languages, the set $\mathbb{Z}_m$ is usually not provided. Instead, $x \bmod y$ is built-in as a functions on *int*.

Booleans

On booleans, we can define the usual boolean operations conjunction (usually written & or &&), disjunction (usually written | or ||), and negation (usually written !).
Moreover, we have the equality and inequality functions, which take two objects x, y and return a boolean. These are usually written $x == y$ and $x! = y$ in text files languages and $x = y$ and $x \neq y$ on paper.

A.5.3 Set Constructors

From the base sets, we build all other sets by applying set constructors. Those are operations that take sets and return new sets.
The following table gives an overview, where we also list the size of each set according to Def. A.33:

the characters of virtually any writing system, many scientific symbols, emojis, etc. Many programming languages assume that there is one character for every primitive integers, e.g., typically 2^{32} characters.

set	description/definition	size
	typical constructors in mathematics	
$A \uplus B$	disjoint union	$\lvert A\rvert + \lvert B\rvert$
$A \times B$	(Cartesian) product	$\lvert A\rvert * \lvert B\rvert$
A^n for $n \in \mathbb{N}$	n-dimensional vectors over A	$\lvert A\rvert^n$
B^A or $A \to B$	functions from A to B	$\lvert B\rvert^{\lvert A\rvert}$
$\mathcal{P}(A)$	power set, equivalent to $\mathbb{B}^A$	$2^{\lvert A\rvert} = \begin{cases} 2^n & \text{if } \lvert A\rvert = n \\ U & \text{otherwise} \end{cases}$
$\{x \in A \mid P(x)\}$	subset of A given by property P	$\leq \lvert A\rvert$
$\{f(x) : x \in A\}$	image of operation f when applied to elements of A	$\leq \lvert A\rvert$
A/r	quotient set for an equivalence relation r on A	$\leq \lvert A\rvert$
	selected additional constructors often used in computer science	
A^*	lists over A	$\begin{cases} 1 & \text{if } A = \varnothing \\ U & \text{if } \lvert A\rvert = U \\ C & \text{otherwise} \end{cases}$
$A^?$	optional element[7] of A	$1 + \lvert A\rvert$
	for new names $l_1, \ldots, l_n$	
$enum\{l_1, \ldots, l_n\}$	enumeration: like $\mathbb{Z}_n$ but also introduces named elements l_i of the enumeration	n
$l_1(A_1) \mid \ldots \mid l_n(A_n)$	labeled union: like $A_1 \uplus \ldots \uplus A_n$ but also introduces named injections l_i from A_i into the union	$\lvert A_1\rvert + \ldots + \lvert A_n\rvert$
$\{l_1 : A_1, \ldots, l_n : A_n\}$	record: like $A_1 \times \ldots \times A_n$ but also introduces named projections l_i from the record into A_i	$\lvert A_1\rvert * \ldots * \lvert A_n\rvert$
inductive data types[8]		C
classes[9]		U

A.5.4 Characteristic Functions of the Set Constructors

Every set constructor comes systematically with characteristic functions into and out of the constructed sets C. These functions allow building elements of C or using elements of C for other computations.

For some sets, these functions do not have standard notations in mathematics. In those cases, different programming languages may use slightly different notations.

The following table gives an overview:

set C	build an element of C	use an element x of C
$A_1 \uplus A_2$	$inj_1(a_1)$ or $inj_2(a_2)$ for $a_i \in A_i$	pattern-matching
$A_1 \times A_2$	(a_1, a_2) for $a_i \in A_i$	$x.i \in A_i$ for $i = 1, 2$
A^n	$(a_1, \ldots, a_n)$ for $a_i \in A$	$x.i \in A$ for $i = 1, \ldots, n$
B^A	$(a \in A) \mapsto b(a)$	$x(a)$ for $a \in A$
A^*	$[a_0, \ldots, a_{l-1}]$[10] for $a_i \in A$	pattern-matching
$A^?$	$None$ or $Some(a)$ for $a \in A$	pattern-matching
$enum\{l_1, \ldots, l_n\}$	l_1 or $\ldots$ or l_n	switch statement or pattern-matching
$l_1(A_1) \mid \ldots \mid l_n(A_n)$	$l_1(a_1)$ or $\ldots$ or $l_n(a_n)$ for $a_i \in A_i$	pattern-matching
$\{l_1 : A_1, \ldots, l_n : A_n\}$	$\{l_1 = a_1, \ldots, l_n = a_n\}$ for $a_i \in A_i$	$x.l_i \in A_i$
inductive data type A	$l(u_1, \ldots, u_n)$ for a constructor l of A	pattern-matching
class A	**new** A	$x.l(u_1, \ldots, u_n)$ for a field l of A

[7] An optional element of A is either absent or an element of A.

[8] These are too complex to define at this point. They are a key feature of functional programming languages like SML.

[9] These are too complex to define at this point. They are a key feature of object-oriented programming languages like Java.

[10] Mathematicians start counting at 1 and would usually write a list of length n as $[a_1, \ldots, a_n]$. However, computer scientists always start counting at 0 and therefore write it as $[a_0, \ldots, a_{n-1}]$. We use the computer science numbering here.

Bibliography

[ALR01] A. Algirdas, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001. University of Newcastle upon Tyne, Computing Science.