Secure and Dependable Systems

Jürgen Schönwälder

June 11, 2019

Abstract

This memo contains annotated slides for the course “Secure and Dependable Systems”.

Contents

I Introduction 3

Motivation 4

Classic Computing Disasters 8

Dependability Concepts and Terminology 15

Dependability Metrics 28

II Software Engineering 33

General Aspects 34

Software Testing 40

Software Specification 47

Software Verification 59

III Concurrency and Distributed Algorithms 82

Concurrency Overview 83

Model of Distributed Algorithms 88

Events, Causality, Logical Clocks 97

Stable Properties and Snapshots 109

Fault Tolerance and Broadcasts 123

Communicating Sequential Processes 136

IV Cryptography 172

Cryptography Primer 173

Symmetric Encryption Algorithms and Block Ciphers 182

Asymmetric Encryption Algorithms 199

Cryptographic Hash Functions 204

Digital Signatures and Certificates 211

Key Exchange Schemes 220

V Secure Communication Protocols 229

Pretty Good Privacy 230

Transport Layer Security 238

Secure Shell 248

VI Information Hiding and Privacy 263

Steganography and Watermarks 264

Covert Channels 273

Anonymization Terminology 278

Mixes and Onion Routing 286

VII Operating System Security 291

Authentication 294

Access Control 295

Part I

Introduction

The aim of this part is to motivate why security and dependability of computing systems are important and to look into recent security failures in order to understand complexities involved in building secure and dependable systems. This part also introduces the terminology and key concepts that are used by the dependability community.

Motivation

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

Can we trust computers?

- How much do you trust (to function correctly)
  - personal computer systems and mobile phones?
  - cloud computing systems?
  - planes, trains, cars, ships?
  - navigation systems?
  - communication networks (telephones, radios, tv)?
  - power plants and power grids?
  - banks and financial trading systems?
  - online shopping and e-commerce systems?
  - social networks and online information systems?
  - information used by insurance companies?
  - ...
- Distinguish between (i) what your intellect tells you to trust and (ii) what you trust in your everyday life.

Computers are so complex and ubiquitous that we have virtually no other choice than trusting hardware and software created by others. We simply cannot verify everything from the ground up even if we would have access to all source code and all hardware specifications. This was very nicely explained by Ken Thompson during his Turing Award lecture [30].

- Stage 1: It is possible to write programs that generate themself. Example:

```
1 char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){printf(f,34,f,34,10);}
```

- Stage 2: A compiler can use knowledge already built into the compiler to produce a new compiler. See this example, which may be part of a C compiler implementing handling of backslash escape sequences.

```
1  char getchar_escaped(void)
2  {
3      char c;
4      if ((c = getchar()) != '\\') return c;
5      switch ((c = getchar())) {
6      case '\\': return '\\';
7      case 'n':  return '\n';
8      default:   return c;
9      }
10 }
```

  The code above never says to which ASCII code point `\n` resolves. This knowledge is already part of the compiler, i.e., this information was added when the compiler was bootstrapped and later removed.

- Stage 3: Modify a compiler to (i) implant a backdoor (if compiling login.c, generate code that allows me to open a backdoor without the need of any credentials) and (ii) implant code that modifies the compiler to reintroduce (i) and (ii) whenever a new compiler is generated. Once there is a new compiler, remove the implanted code.

The result is a compiler that generates backdoors with no trace of this backdoor in the source code. People verifying all source code will come to the false conclusion that there is no backdoor. Ken Thompson concludes: "You can't trust code that you did not totally create yourself". In particular, you cannot trust machine code even if you have read all source code needed to produce the machine code. A compromised compiler can compromise your linker, your debugger, your disassembler, and all your tools to analyze software. Imagine what happens if your CPU is compromised...

Importance of Security and Dependability

- Software development processes are often too focused on functional aspects and user interface aspects (since this is what sells products).
- Aspects such as reliability, robustness against failures and attacks, long-term availability of the software and data, integrity of data, protection of data against unauthorized access, etc. are often not given enough consideration.
- Software failures can not only have significant financial consequences, they can also lead to environmental damages or even losses of human lifes.
- Due to the complexity of computing systems, the consequences of faults in one component are very difficult to estimate.
- Security and dependability aspects must be considered during all phases of a software development project.

This cannot be stressed enough:

> **Once you leave university and you work as a professional software developer (i.e., you stop writing throw-away toy programs), you start having responsibility for the software you produce since others will trust your software.**

In general, you will not be able to decide whether your software will be used in a context where failures can have substantial or even catastrophic consequences. Hence, you carry a lot of responsibility whenever you produce software and you need to remind yourself of this responsibility regularly. And you will find yourself in situation where you have to defend that producing “good” software is important and ultimately in the interest of whoever finances the project you are working on. There is a lot to be said about ethics and computer science.

Classic Computing Disasters

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

IoT Remove Control Light Bulbs 2018

The light bulb (and other IoT smart home products such as remote controlled sockets) use an Espressif ESP 8266 system on a chip. The software on the light bulb was found to get quite a few things wrong:

- It stores credentials in plaintext in flash memory that can be read with relatively little effort (even after the bulb has been thrown away).
- It uses a software update mechanism that can be tricked with some minor effort to load compromised firmware (firmware is not signed).
- It leaks unnecessary sensitive information to a cloud platform.
- HTTP traffic is largely not encrypted, but MQTT traffic is encrypted. Message Queuing Telemetry Transport (MQTT) is a publish-subscribe-based messaging protocol commonly used between IoT devices and cloud servers. The implementation uses relatively weak cryptographic algorithms and has key management issues.
- It uses a mechanism to learn WLAN credentials from a smart phone that can leak credentials to any observers.

It is possible to buy these light bulbs from Amazon for roughly 15 Euros. The Amazon product description (last checked 2019-02-06) includes an image that shows a wireless symbol followed by the text “Remove Control”. They seem to be honest in their product advertisement.

The manufacturer Tuya has announce in early 2019 that they will fix the software.

For further information:

- https://media.ccc.de/v/35c3-9723-smart_home_-_smart_hack

Spectre: Vulnerability of the Year 2018

```
unsigned char array1[16]            /* base array */
unsigned int array1_size = 16;      /* size of the base array */
int x;                              /* the out of bounds index */
unsigned char array2[256 * 256];    /* instrument for timing channel attack */

// ...

if (x < array1_size) {
    y = array2[array1[x] * 256];
}
```

- Is the above code a vulnerability?

The code seems to be harmless. Well it is not really. But we can easily make it harmless. But even then, is it harmless?

Spectre: Main Memory and CPU Memory Caches

- Memory in modern computing systems is layered:
- Main memory is large but relatively slow compared to the speed of the CPUs
- CPUs have several internal layers of memory caches, each layer faster but smaller
- CPU memory caches are not accessible from outside of the CPU
- When a CPU instruction needs data that is in the main memory but not in the caches, then the CPU has to wait quite a while...

Spectre: Timing Side Channel Attack

- A side-channel attack is an attack where information is gained from the physical implementation of a computer system (e.g., timing, power consumption), rather than weaknesses in an implemented algorithm itself.
- A timing side-channel attack infers data from timing observations.
- Even though the CPU memory cache cannot be read directly, it is possible to infer from timing observations whether certain data resides in a CPU memory cache or not.
- By accessing specific uncached memory locations and later checking via timing observations whether these locations are cached, it is possible to communicate data from the CPU using a cache timing side channel attack.

Side-channel attacks can use many different kinds of side channels. Some examples:

- The size of network packets can reveal which resources are accessed on a web server even if the communication is encrypted.
- The power consumption of displays can reveal what kind of content is displayed.
- Data transmitted over copper wires creates magnetic fields around the wire that can induce a signal on other wires that reveal the original data.
- The variation of power consumption of CPUs has been used to gain information about keys used during cryptographic calculations.

Spectre: Speculative Execution

- In a situation where a CPU would have to wait for slow memory, simply guess a value and continue excution speculatively; be prepared to rollback the speculative computation if the guess later turns out to be wrong; if the guess was correct, commit the speculative computation and move on.
- Speculative execution is in particular interesting for branch instructions that depend on memory cell content that is not found in the CPU memory caches
- Some CPUs collect statistics about past branching behavior in order to do an informed guess. This means we can train the CPUs to make a certain guess.
- Cache state is not restored during the rollback of a speculative execution.

The fact that the CPU internal cache state is not restored during the rollback is being exploited by Spectre. Of course, CPUs could be "fixed" to restore the cache state as well but this would be very costly to implement and hence may defeat the advantage gained by speculative execution.

Spectre: Reading Arbitrary Memory

- Algorithm:
  1. create a small array `array1`
  2. choose an index `x` such that `array1[x]` is out of bounds
  3. trick the CPU into speculative execution (make it to read `array1_size` from slow memory and to guess wrongly)
  4. create another uncached memory array called `array2` and read `array2[array1[x]]` to load this cell into the cache
  5. read the entire `array2` and observe the timing; it will reveal what the value of `array1[x]` was
- This could be done with JavaScript running in your web browser; the first easy "fix" was to make the JavaScript time API less precise, thereby killing the timing side channel.

Spectre is exploiting a design problem in modern CPUs. There is no easy fix since the root cause is your hardware. A lot of work was spent in 2018 to harden systems such that it is getting difficult to exploit the problem residing in the design of modern CPUs.

For further information:

- P. Kocher et al.: Spectre Attacks: Exploiting Speculative Execution
- M. Lipp et al.: Meltdown
- https://www.youtube.com/watch?v=6O8LTwVfTVs
- https://www.youtube.com/watch?v=mgAN4w7LH2o

Dependability Concepts and Terminology

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

Dependability is a very general but important concept when we talk about computing systems. In this part, we define the basic concepts and the terminology, following [3]. Note that [3] provides a much more detailed treatment of the topic and students are encouraged to read the entire paper in order to learn more about fault and failure classifications and fault tolerance techniques.

System and Environment and System Boundary

Definition (system, environment, system boundary)

A *system* is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. The other systems are the *environment* of the given system. The *system boundary* is the common frontier between the system and its environment.

- Note that systems almost never exist in isolation.
- We often forget to think about all interactions of a system with its environment.
- Well-defined system boundaries are essential for the design of complex systems.

An example for a system could be the standard C library. The system boundary of the C library is defined by the set of C library calls. The functional specification of the C library calls is the C language standard. The C library implementation may use other libraries (components) and it uses other systems (e.g., the operating system kernel) that are part of the C library's runtime environment.

Similarly, the operating system kernel can be seen as a system as well. The set of operating system calls forms the system boundary. The operating system uses other systems such as hardware components or other integrated hardware and software components that are attached to a computer.

It is crucial to think about systems, their environments and their dependencies. Catastrophic failures are sometimes caused by an uncontrolled propagation of failures from one system to another. For example, denial of service attacks can be more effective if attackers find systems that amplify their attacks and/or make it difficult to trace back where the attack originated from.

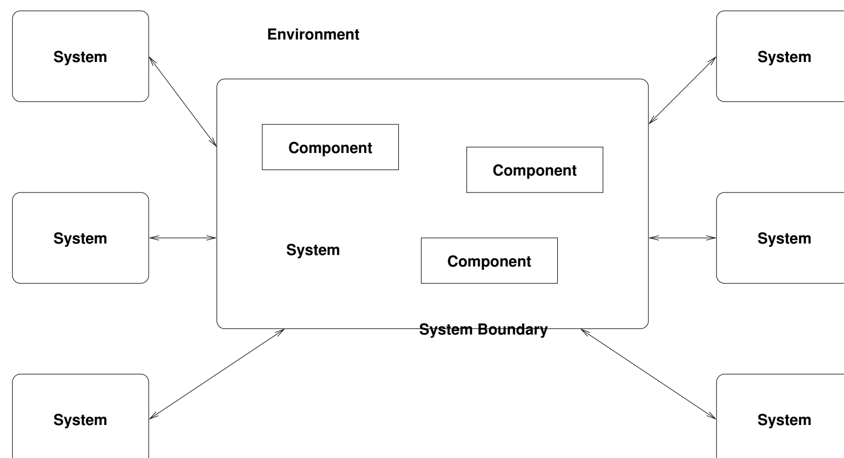
Components and State

Definition (components)

The structure of a system is composed out of a set of *components*, where each component is another system. The recursion stops when a component is considered atomic.

Definition (total state)

The *total state* of a given system is the set of the following states: computation, communication, stored information, interconnection, and physical condition.



Function and Behaviour

Definition (function and functional specification)

The *function* of a system is what the system is intended to do and is described by the *functional specification*.

Definition (behaviour)

The *behaviour* of a system is what the system does to implement its function and is described by a sequence of states.

It is important to stress that a functional specification is required when we talk about correctness. Without a clear and complete functional specification, we can not decide whether a system behaves correctly or not.

We will look into program verification techniques later. These techniques verify the correctness of a program against a functional specification. If the functional specification is incorrect, then of course the verified program can be seen as incorrect, even though it is correct regarding the incorrect functional specification.

Since functional specifications are often not formalized, it is in practice often necessary to derive a formalized functional specification out of the original more informal functional specification in order to apply program verification techniques. This formalization step can (i) introduce faults that did not exist in the original informal functional specification or (ii) slightly change the specification such that it differs in some subtle aspects from the original more informal functional specification.

Mistakes in functional specifications are often very expensive to fix. One reason is that they are often detected late in the software development process, for example at system integration time or at deployment time or when the software is already in production.

Service and Correct Service

Definition (service)

The *service* delivered by a system is its behaviour as it is perceived by a its user(s); a user is another system that receives service from the service provider.

Definition (correct service)

Correct service is delivered when the service implement the system function.

Recall that the system function is defined in the functional specification. If the functional specification is incomplete (a very likely scenario for many systems), then the service provided can be undefined in certain situations, i.e., it is neither correct nor incorrect.

Failure versus Error versus Fault

Definition (failure)

A *service failure*, often abbreviated as *failure*, is an event that occurs when the delivered service deviates from correct service.

Definition (error)

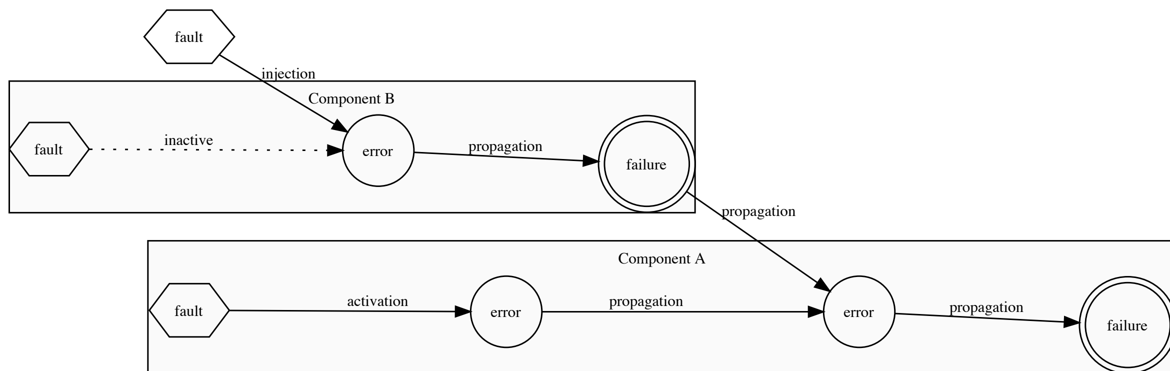
An *error* is the part of the total state of the system that may lead to its subsequent service failure.

Definition (fault)

A *fault* is the adjudged or hypothesized cause of an error. A fault is *active* when it produces an error, otherwise it is *dormant*.

The dependability community defines the terms fault, error, and failure with a clear distinction between them. Other communites are less precise about the usage of these terms and they may even have an entirely different terminology in place. For example, the software engineering community refers to faults often as bugs. (Some people claim the term bug goes back to computers that used mechanical relais and one of them stopped working because of a bug trapped in a relay.)

Note that the definitions imply an error propagation model:



It is follows directly from this graph that it is desirable

- to avoid faults and to reduce faults, and
- to prevent dormant faults from getting activated, and
- to detect and handle errors so that they do not propagate, and
- to detect and handle failures of other components or systems, and
- to reduce the number of ways external phenoma can inject faults.

It further follows directly that any program consuming data from an external source must carefully check that the data matches the expectations of the program. So called SQL injection attacks exploit programs that fail to carefully validate the input and as a consequence send unexpected SQL queries to a database system. Fuzzying techniques try to inject errors by generating random input that is likely to trigger errors.

Dependability

Definition (dependability - original)

Dependability is the ability of a system to deliver service than can justifiably be trusted.

Definition (dependability - revised)

Dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.

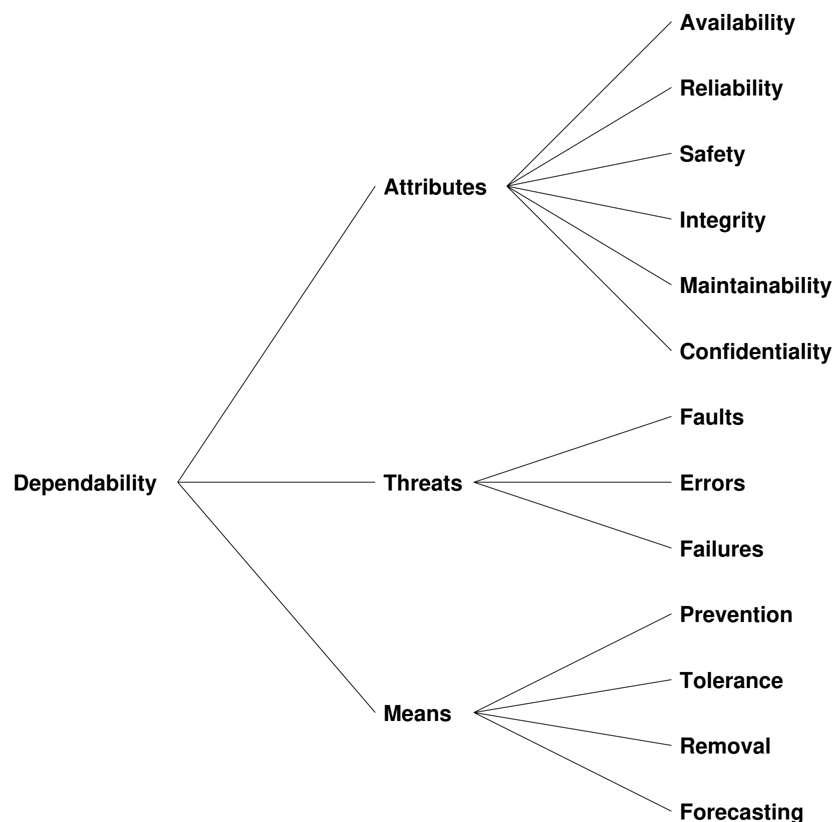
- The revised definition provides a criterion for deciding if a system is dependable.
- Trust can be understood as a form of accepted dependance.

Dependability Attributes

Definition (dependability attributes)

Dependability has the following attributes:

- *Availability*: readiness to deliver correct service
- *Reliability*: continuity of correct service
- *Safety*: absence of catastrophic consequences on the user(s) and the environment
- *Integrity*: absence of improper system alterations
- *Maintainability*: ability to undergo modifications and repairs
- *Confidentiality*: absence of unauthorized disclosure of information



Dependability and Security

Definition (security)

Security is a composite of the attributes of confidentiality, integrity, and availability.

- The definition of dependability considers security as a subfield of dependability. This does, however, not reflect how research communities have organized themselves.
- As a consequence, terminology is generally not consistent. Security people, for example, talk about vulnerabilities while dependability people talk about dormant faults.

Fault Prevention

Definition (fault prevention)

Fault prevention aims at preventing the occurance or introduction of faults.

- Application of good software engineering techniques and quality management techniques during the entire development process.
- Hardening, shielding, etc. of physical systems to prevent physical faults.
- Maintenance and deployment procedures (e.g., firewalls, installation in access controlled rooms, backup procedures) to prevent malicious faults.

Fault prevention is a core topic of software engineering. The selection of a proper programming language for a given task can have a big impact on the number and kind of faults that can be produced. For example, a programming language that does automatic bounds checking for memory objects dramatically reduces buffer overrun faults. Similarly, a programming language that does automatic memory management dramatically reduces problems due to memory leaks or the usage of deallocated memory.

In some parts of the industry, subsets of programming general purpose programming languages are used. An example is MISRA-C, which is essentially a collection of coding standards for safety-critical systems. MISRA-C originated from the automotive industry but is meanwhile used also in other contexts.

Another important aspect is system complexity. Complex systems are very hard to maintain and extend without introducing faults as side effects. It is thus crucial to find good system designs and abstractions that encourage high cohesion and loose coupling. And it is crucial to maintain a good system design (or even to improve the system design) during the lifecycle of a software product.

Fault Tolerance

Definition (fault tolerance)

Fault tolerance aims at avoiding service failures in the presence of faults.

- Error detection aims at detecting errors that are present in the system so that recovery actions can be taken.
- Recovery handling eliminates errors from the system by rollback to an error-free state or by error compensation (exploiting redundancy) or by rollforward to an error-free state.
- Fault handling prevents located faults from being activated again.

With the decreasing cost for computing power, it is meanwhile feasible to use replication of data and computations in order to produce redundancy that can be used to compensate errors and failures. For example, a query sent to a search engine may be given to multiple independent backend systems and the first response that is returned by the backends is returned to the user. This not only provide fast response times but also handles occasional failures of backend systems nicely.

Data replication is another enabler for fault tolerance. Storage systems use replication at the system level, across systems in a computing center, and even across entire computing centers. Of course, guaranteeing data consistency in a distributed system with replicated data is not trivial. In order to be efficient, modern systems often work with update semantics that are not atomic but only eventually consistent. In other words, one can view the entire system as always being converging to an ideal consistent state that might never be reached.

That said, there have also been examples where fault tolerance mechanisms, due to their complexity, have caused failures that otherwise would not have occured.

Examples:

- RAID disk arrays store redundant information in order to tolerate disk failures. This is a form of redundant data storage, which also can increate throughput.
- Online systems like Google may process a received query N times and return the first response. This is a form of redundant computing, which also increases response time.

Fault Removal

Definition (fault removal)

Fault removal aims at reducing the number and severity of faults.

- Fault removal during the development phase usually involves verification checks whether the system satisfies required properties.
- Fault removal during the operational phase is often driven by errors that have been detected and reported (corrective maintenance) or by faults that have been observed in similar systems or that were found in the specification but which have not led to errors yet (preventive maintenance).
- Sometimes it is impossible or too costly to remove a fault but it is possible to prevent the activation of the fault or to limit the possible impact of the fault, i.e, its severity.

Fault removal during the development phase is most effective. Modern software engineering techniques therefore encourage continued and extensive testing. Some modern development practices require developers to write collections of test cases before starting to write the actual code. Furthermore, quality control at later stages of the development process often involves people who were not involved in the code development itself. Furthermore, the selection of programming languages, the training of programmers, the programming paradigms used and so on all have an influence on the quality of the produced software.

Fault removal in software systems during the operational phase is often done by installing updates or patches. Software that is used in an open environment must be patched regularly. Patch management is the process of using a strategy and plan of what patches should be applied to which systems at a specified time. In other words, producers of products that include software must have a plan how to provide and distribute patches over the entirely lifecycle of the products. Similarly, users of products that include software must have a plan who is responsible to keep products patched.

Automatic software update mechanisms have emerged in the past few years in order to reduce the burden on the user side and to improve the user experience. However, we are far from having robust automatic software update mechanisms widely deployed, in particular considering embedded systems.

Fault Forecasting

Definition (fault forecasting)

Fault forecasting aims at estimating the present number, the future incidence, and the likely consequences of faults.

- Qualitative evaluation identifies, classifies, and ranks the failure modes, or the event combinations that would lead to failures.
- Quantitative evaluation determines the probabilities to which some of the dependability attributes are satisfied.

Fault forecasting is often done by collecting statistics about the changes made to a software system and the number of bugs reported and fixed over time. The idea is to be able to predict how stable a program is or how long one should wait after the release of a major new version until most of the faults have been found and removed.

A recent study of open source computer network control software came to the conclusion that network operators should wait almost a year before deploying a major new release.

Dependability Metrics

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

There are some metrics that measure reliability, availability, and safety dependability attributes. However, there are no commonly accepted metrics for correctness or security attributes.

Reliability and MTTF/MTBF/MTTR

Definition (reliability)

The *reliability* $R(t)$ of a system S is defined as the probability that S is delivering correct service in the time interval $[0, t]$.

- A metric for the reliability $R(t)$ for non repairable systems is the Mean Time To Failure (MTTF), normally expressed in hours.
- A metric for the reliability $R(t)$ for repairable systems is the Mean Time Between Failures (MTBF), normally expressed in hours.
- The mean time it takes to repair a repairable system is called the Mean Time To Repair (MTTR), normally expressed in hours.
- These metrics are meaningful in the steady-state, i.e., when the system does not change or evolve.

Availability

Definition (availability)

The *availability* $A(t)$ of a system S is defined as the probability that S is delivering correct service at time t.

- A metric for the average, steady-state availability of a repairable system is $A = MTBF/(MTBF + MTTR)$, normally expressed in percent.
- A certain percentage-value may be more or less useful depending on the "failure distribution" (the "burstiness" of the failures).
- Critical computing systems usually have to guarantee a certain availability. Availability requirements are usually defined in service level agreements.

The Amazon Service Level Agreement says (retrieved 2019-02-07):

> AWS will use commercially reasonable efforts to make the Included Products and Services each available with a Monthly Uptime Percentage (defined below) of at least 99.99%, in each case during any monthly billing cycle (the "Service Commitment"). In the event any of the Included Products and Services do not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

The Google Cloud Filestore Service Level Agreement for says (retrieved 2019-02-07):

> During the term of the Google Cloud Platform License Agreement or Google Cloud Platform Reseller Agreement (as applicable, the "Agreement"), the Covered Service will provide a Monthly Uptime Percentage to Customer of at least 99.9% (the "Service Level Objective" or "SLO").

The Jacobs University CampusNet Server Service (retrieved 2019-02-07):

> Service Reliability
>
> The HTTPS service is 99% reliable, calculated per month.

Our IT service is using a different terminology (perhaps a German to English translation issue).

Availability and the "number of nines"

Availability	Downtime per year	Downtime per month	Downtime per week	Downtime per day
90%	36.5 d	72 h	16.8 h	2.4 h
99%	3.65 d	7.20 h	1.68 h	14.4 min
99.9%	8.76 h	43.8 min	10.1 min	1.44 min
99.99%	52.56 min	4.38 min	1.01 min	8.64 s
99.999%	5.26 min	25.9 s	6.05 s	864.3 ms
99.9999%	31.5 s	2.59 s	604.8 ms	86.4 ms

- It is common practice to express the degrees of availability by the number of nines. For example, "5 nines availability" means 99.999% availability.

Note that increased availability comes with additional costs. So there needs to be a business case. David Stephens wrote wrote on a blog (I did not invest time to verify this so take this is just that, a blog post):

> A study by Compuware and Forrester Research in 2011 found an average business cost of US$14,000 per minute for mainframe outages. Using this figure, outages for systems with five-nines cost about US$73,500 per year. For systems with four-nines (99.99%, or 52.56 minutes downtime) this increases to US$735,000, and three-nines (99.9%, or 525.6 minutes) US$7.3 million. So a business case to improve availability from four to five-nines needs the extra hardware, software and resources to cost less than about US$660,000 per year. When we're talking about mainframe hardware and software, $660,000 doesn't buy much. A jump from three-nines to four-nines may save millions, and is a far easier business case.

Safety

Definition (safety)

The *safety* $S(t)$ of a system S is defined as the probability that S is delivering correct service or has failed in a manner that does cause no harm in $[0, t]$.

- A metric for safety $S(t)$ is the Mean Time To Catastrophic Failure (MTTC), defined similarly to MTTF and normally expressed in hours.
- Safety is reliability with respect to malign failures.

Fail-safe systems are systems that have been engineered such that in the event of a specific type of failure, the system inherently responds in a way that will cause no or minimal harm to other equipment, the environment or to people.

Part II

Software Engineering

This part only touches on some aspects of software engineering since we have a separate course on software engineering. In general, software engineering is a highly important topic in the commercial and even the research world, but also very difficult to teach since students lack an understanding what it means to work on really large software projects and within financial and time constraints.

In the following, we will first focus on topics that are relevant for preventing or detecting faults before software become deployed for production. We will then look at techniques to specify the correctness of programs and to verify whether a program is correct. We will also look into ways to describe and verify patterns of interaction in concurrent systems.

The treatment of Floyd-Hoare triples and Floyd-Hoare logic is largely based on Mike Gordon's execellent "Background reading on Hoare Logic".

General Aspects

8 General Aspects

9 Software Testing

10 Software Specification

11 Software Verification

Definitions of Software Engineering

Definition

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. (IEEE Standard Glossary of Software Engineering Terminology)

Definition

The establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines. (Fritz Bauer)

Definition

An engineering discipline that is concerned with all aspects of software production. (Ian Sommerville)

Good engineering essentially implies to produce a technical artefact that meets its technical requirements within a given budget and time constraint. The means are structured development processes.

Good Software Development Practices

- Coding Styles
- Documentation
- Version Control Systems
- Code Reviews and Pair Programming
- Automated Build and Testing Procedures
- Issue Tracking Systems

- Coding Styles

  Readability is key. Since code is in general written and maintained by multiple people, it is helpful to agree on a common coding style. Good program development environments help to follow a common coding style. There are coding styles that were designed to minimize programming errors (e.g., MISRA C[1]).

- Documentation

  Documentation explaining details not obvious from the code is important. Nowadays, documentation is often generated by tools (e.g., doxygen[2]) from structured documentation comments included inline in the source code. (Motivation: Keep code and documentation consistent.)

- Version Control Systems

  Version control systems such as git[3] help to track different versions of a code base and they support distributed and loosely coupled development schemes.

- Code Reviews and Pair Programming

  Peer review of source code helps to improve the quality of the code committed to a project and it facilitates peer-learning in a software development team. An extreme form is pair programming where coding is always done in pairs of two programmers.

- Automated Build and Testing Procedures

  The software build and testing process should be fully automated. Automated builds on several target platforms and the automated execution of regression tests triggered by a commit to a version control system.

- Issue Tracking Systems

  Issue tracking systems organize the resolution of problems and feature requests. All discussions related to a software issue are recorded and archived in a discussion thread. Issues are usually labeled with metadata, which allows development managers to collect insights about the software production process.

[1] https://en.wikipedia.org/wiki/MISRA_C
[2] https://en.wikipedia.org/wiki/Doxygen
[3] https://en.wikipedia.org/wiki/Git

Choice of Programming Languages

- Programming languages serve different purposes and it is important to select a language that fits the given task
- Low-level languages can be very efficient but they tend to allow programmers to make more mistakes
- High-level languages and in particular functional languages can lead to very abstract but also very robust code
- Concurrency is important these days and the mechanisms available in different programming languages can largely impact the robustness of the code
- Programming languages must match the skills of the developer team; introducing a new languages requires to train developers
- Maintainability of code must be considered when programming languages are selected

Beware of “if all you have is a hammer, then everything starts to look like a nail” and be open to learn/use different languages and frameworks. And since code needs maintenance, it may not be the best choice to use a programming language that itself is not yet stable or to use a programming language where there is little expertise available to maintain the code. There is also a time dimension; sometimes it can be desirable to get a first version done in a language that supports prototyping well and to be prepared to rewrite core components later if the software product is successful. Hence, the selection of a programming language itself is an engineering decision where trade-offs have to be considered and weighted. As a professional, you should stay away from “religious debates” about programming languages.

Defensive Programming

- It is common that functions are only partially defined.
- Defensive programming requires that the preconditions for a function are checked when a function is called.
- For some complex functions, it might even be useful to check the postcondition, i.e., that the function did achieve the desired result.
- Many programming languages have mechanisms to insert assertions into the source code in order to check pre- and postconditions.

C programmer can use the `assert()` macro defined in `assert.h` to add assertions to their code. If the assertion (an expression) fails, then diagnostic messages are written to the standard error. The evaluation of assert expressions can be disabled to save execution time once the program is well debugged. Note that assertions should be used to detect programming errors (i.e., function calls in an invalid context), they are not to be used to handle runtime exceptions.

```
1  #include <assert.h>
2
3  int average(int *a, int size)
4  {
5      int sum = 0;
6      assert(a && size > 0);
7      for (int i = 0; i < size; i++) {
8          sum += a[i];
9      }
10     return sum / size;
11 }
```

Another example showing that sometimes testing whether a certain function has worked correctly is easier than implementing the complex function itself. And such tests may even be reused in other parts of a program:

```
1  #include <assert.h>
2
3  void sort(int *a, size_t n)
4  {
5      assert(a);
6      recursive_super_duper_sort(a, 0, n);
7      assert(is_sorted(a, n));
8  }
9
10 static bool is_sorted(int const *a, size_t n)
11 {
12     for (size_t i=0; i<n-1; i++) {
13         if (a[i] > a[i+1]) { return false; }
14     }
15     return true;
```

```
16  }
17
18  int binary_search(int *a, size_t n, int value)
19  {
20      assert(a && is_sorted(a, n));
21      // ...
22  }
```

Software Testing

8 General Aspects

9 Software Testing

10 Software Specification

11 Software Verification

Unit and Regression Testing

- Unit testing
  - Testing of units (abstract data types, classes, ...) of source code.
  - Usually supported by special unit testing libraries and frameworks.
- Regression testing
  - Testing of an entire program to ensure that a modified version of a program still handles all input correctly that an older version of a program handled correctly.
- A software bug reported by a customer is primarily a weakness of the regression test suite.
- Modern agile software development techniques rely on unit testing and regression testing techniques.

Resist the temptation to test manually from the command line. Automate testing from the start. And ideally, start coding with writing test cases before you do anything else. This takes discipline but will change the way you approach problems. By constructing test cases before you code, you will focus on understanding the entire problem early on and this helps you to also spot corner cases (where often the specification given to you is not precise enough).

Example: You are asked to implement a function that returns the greatest common divisor (gcd) and a function that returns the lowest common multiple (lcm) of two integer numbers. What are suitable test cases?

Some open source unit testing frameworks:

- check unit testing framework for C
- cmocka unit testing framework for C
- Catch[2] automated test framework for C++
- Google Test Google's C++ test framework
- junit Java unit testing framework that inspired many other testing frameworks

Test Coverages

- The test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs.
- Function coverage:
  - Has each function in the program been called?
- Statement coverage:
  - Statement coverage: Has each statement in the program been executed?
- Branch coverage:
  - Has each branch of each control structure been executed?
- Predicate coverage:
  - Has each Boolean sub-expression evaluated both to true and false?

A common approach to obtain test coverage information is to instrument a program at compilation time (using special compiler options) so that coverage information is collected when the program is executed. The collected raw information can then subsequently be analyzed to obtain coverage reports.

Mutation Testing

- Mutation testing evaluates the effectiveness of a test suite.
- The source code of a program is modified algorithmically by applying mutation operations in order to produce mutants.
- A mutant is "killed" by a test suite if tests fail for the mutant. Mutants that are not "killed" indicate that the test suite is incomplete.
- Mutation operators often mimic typical programming errors:
  - Statement deletion, duplication, reordering, ...
  - Replacement of arithmetic operations with others
  - Replacement of boolean operations with others
  - Replacement of comparison relations with others
  - Replacement of variables with others (of the same type)
- The mutation score is the number of mutants killed normalized by the number of mutants.

Mutation testing tools usually operate on the abstract syntax tree representation of a program in order to generate mutants. Since mutation testing is computationally more expensive than collecting coverage statistics for estimating the quality a test suite, finding ways to generate "good" mutants (i.e., mutants that have a high probability to not be killed) is an important factor to scale up mutation testing for large code bases.

Fuzzying

- Fuzzying or fuzz testing feeds invalid, unexpected, or simply random data into computer programs.
  - Some fuzzers can generate input based on their awareness of the structure of input data.
  - Some fuzzers can adapt the input based on their awareness of the code structure and which code paths have already been covered.
- The "american fuzzy lop" (AFL) uses genetic algorithms to adjust generated inputs in order to quickly increase code coverage.
- AFL has detected a significant number of serious software bugs.

Running AFL is relatively simple.

- First, the source has to be compiled with a special version of `gcc` (or `clang`) typically called `afl-gcc` (or `afl-clang`) to instrument the generated code to collect coverage information.
- Assuming the program can read input from a file, a directory is created with a number of files with valid input. These input files provide the starting point for the ALF fuzzer.
- The `afl-fuzz` program is run to generate new inputs that are fed into the instrumented program. The coverage information collected while executing the instrumented program is used to optimize the generation of random inputs.
- Once a program exits abnormally or it hangs or it uses more than a predefined amount of memory, the test cases triggering the problem are saved in order to allow someone to investigate the problem and to fix the code.

Fuzzying is amazingly effective in finding problems in parsers, i.e., the parts of a program that read specific file formats. Fuzzying has also been used successfully to generate random messages that can be sent over the Internet to a system under test (fuzzying communication protocols).

Fault Injection

- Fault injection techniques inject faults into a program by either
  - modifying source code (very similar to mutation testing) or
  - injecting faults at runtime (often via modified library calls).
- Fault injection can be highly effective to test whether software deals with rare failure situations, e.g., the injection of system calls failures that usually work.
- Fault injection can be used to evaluate the robustness of the communication between programs (deleting, injecting, reordering messages).
- Can be implemented using library call interception techniques.

A Linux fault injection library is `libfiu`. It comes with wrappers for POSIX system calls that can be used to fail system calls with certain percentages.

```
fiu-run -x -c "enable_random name=posix/io/rw/read,probability=0.05" fortune
```

There are also tools to enable/disable injected faults by an external program.

```
fiu-run -x top
fiu-ctrl -c "enable name=posix/io/oc/open" `pidof top`
fiu-ctrl -c "disable name=posix/io/oc/open" `pidof top`
```

Multiple Independent Computations

- Dionysius Lardner 1834:

  The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods.

- Charles Babbage, 1837:

  When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, we may then be quite sure of the accuracy of them all.

Safety-relevant systems are sometimes constructed such that they do independent computations on different hardware systems and there is a fail-safe (or proven to be correct) unit comparing the independently computed results. Note that this also applies to the software used. In order to be able to detect errors, it is possible to let two independent teams implement the same functionality using different programming languages and algorithms.

Software Specification

8 General Aspects

9 Software Testing

10 Software Specification

11 Software Verification

Formal Specification and Verification

Definition (formal specification)

A *formal specification* uses a formal (mathematical) notation to provide a precise definition of what a program should do.

Definition (formal verification)

A *formal verification* uses logical rules to mathematically prove that a program satisfies a formal specification.

- For many non-trivial problems, creating a formal, correct, and complete specification is a problem by itself.
- A bug in a formal specification leads to programs with verified bugs.

Floyd-Hoare Triple

Definition (hoare triple)

Given a state that satisfies precondition P, executing a program C (and assuming it terminates) results in a state that satisfies postcondition Q. This is also known as the "Hoare triple":

$$\{P\}\ C\ \{Q\}$$

- Invented by Charles Anthony ("Tony") Richard Hoare with original ideas from Robert Floyd (1969).
- Hoare triple can be used to specify what a program should do.
- Example:

$$\{X = 1\}\ X := X + 1\ \{X = 2\}$$

The classic publication introducing Hoare logic is [13]. Tony Hoare has made several other notable contributions to computer science: He invented the basis of the Quicksort algorithm (published in 1962) and he has developed the formalism Communicating Sequential Processes (CSP) to describe patterns of interaction in concurrent systems (published in 1978).

P and Q are conditions on program variables. They will be written using standard mathematical notation and logical operators. The predicate P defines the subset of all possible states for which a program C is defined. Similarly, the predicate Q defines the subset of all possible states for which the program's result is defined.

It is possible that different programs satisfy the same specification:

$$\{X = 1\}\ Y := 2\ \{X = 1 \land Y = 2\}$$

$$\{X = 1\}\ Y := 2 * X\ \{X = 1 \land Y = 2\}$$

$$\{X = 1\}\ Y := X + X\ \{X = 1 \land Y = 2\}$$

Partial Correctness and Total Correctness

Definition (partial correctness)

An algorithm starting in a state that satisfies a precondition P is *partially correct with respect to* P *and* Q if results produced by the algorithm satisfy the postcondition Q. Partial correctness does not require that always a result is produced, i.e., the algorithm may not always terminate.

Definition (total correctness)

An algorithm is *totally correct with respect to* P *and* Q if it is partially correct with respect to P and Q and it always terminates.

The distinction between partial correctness and total correctness is of fundamental importance. Total correctness requires termination, which is generally impossible to prove in an automated way as this would require to solve the famous halting problem. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

A definition of the form $\{P\}\ C\ \{Q\}$ usually provides a partial correctness specification. We may use the notation $[P]\ C\ [Q]$ for a total correctness specification.

Hoare Notation Conventions

1. The symbols V, V_1, ..., V_n stand for arbitrary variables. Examples of particular variables are X, Y, R etc.
2. The symbols E, E_1, ..., E_n stand for arbitrary expressions (or terms). These are expressions like $X + 1$, $\sqrt{2}$ etc., which denote values (usually numbers).
3. The symbols S, S_1, ..., S_n stand for arbitrary statements. These are conditions like $X < Y$, $X^2 = 1$ etc., which are either true or false.
4. The symbols C, C_1 , ..., C_n stand for arbitrary commands of our programming language; these commands are described in the following slides.

- We will use lowercase letters such as x and y to denote auxiliary variables (e.g., to denote values stored in variables).

We are focusing in the following on a purely imperative programming model where a global set of variables determines the current state of the computation. A subset of the variables are used to provide the input to an algorithm and another subset of the variables provides the output of an algorithm.

Note that we talk about a programming language consisting of commands and we use the term statements to refer to conditions. This may be a bit confusing since programming languages often call our commands statements and they may call our statements conditions.

Hoare Assignments

- Syntax: $V := E$
- Semantics: The state is changed by assigning the value of the term E to the variable V . All variables are assumed to have global scope.
- Example: $X := X + 1$

Hoare Skip Command

- Syntax: *SKIP*
- Semantics: Do nothing. The state after execution the command *SKIP* is the same as the state before executing the command *SKIP*.
- Example: *SKIP*

The $SKIP$ command does nothing. It is still useful since it allows us to construct a single conditional command.

Hoare Command Sequences

- Syntax: $C_1; \ldots; C_n$
- Semantics: The commands $C_1, \ldots, C_n$ are executed in that order.
- Example: $R := X; X := Y; Y := R$

The example sequence shown above swaps the content of X and Y. Note that it has a side-effect since it also assigns the initial value of X to R. A specification of the swap program as a Floyd-Hoare triple would be the following:

$$\{X = x \wedge Y = y\}\ R := X; X := Y; Y := R\ \{X = y \wedge Y = x\}$$

Since the program does not involve any loops, it is easy to see that we could also easily specify total correctness:

$$[X = x \wedge Y = y]\ R := X; X := Y; Y := R\ [X = y \wedge Y = x]$$

Hoare Conditionals

- Syntax: *IF S THEN C_1 ELSE C_2 FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Example: *IF $X < Y$ THEN $M := Y$ ELSE $M := X$ FI*

Note that we can use $SKIP$ to create conditional statements without a $THEN$ or $ELSE$ branch:

IF S THEN C ELSE SKIP FI

IF S THEN SKIP ELSE C FI

Hoare While Loop

- Syntax: *WHILE S DO C OD*
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- Example: $WHILE\ \neg(X = 0)\ DO\ X := X - 2\ OD$

Our notation uses a convention that was popular in the 1970s to denote the end of a programming language construct by repeating a keyword with the letters reversed. An early programming language using this notation was Algol 68. You find similar syntactic ideas in Bourne shells (if / fi, case / esac).

Termination can be Tricky

```
1: function COLLATZ(X)
2:     while X > 1 do
3:         if (X%2) ≠ 0 then
4:             X ← (3 · X) + 1
5:         else
6:             X ← X/2
7:         end if
8:     end while
9:     return X
10: end function
```

- Collatz conjecture: The program will eventually return the number 1, regardless of which positive integer is chosen initially.

This program calculates the so called Collatz sequence. The Collatz conjecture is that no matter what value of $n \in \mathbb{N}$ you start with, the sequence will always reach 1. For example, starting with $n = 12$, one gets the sequence $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$.

For further information:

- https://en.wikipedia.org/wiki/Collatz_conjecture

Specification can be Tricky

- Specification for the maximum of two variables:

$$\{\mathbf{T}\}\ C\ \{Y = max(X, Y)\}$$

- C could be:

```
IF X > Y THEN Y := X ELSE SKIP FI
```

- But C could also be:

```
IF X > Y THEN X := Y ELSE SKIP FI
```

- And C could also be:

```
Y := X
```

- Use auxiliary variables x and y to associate Q with P:

$$\{X = x \wedge Y = y\}\ C\ \{Y = max(x, y)\}$$

Obviously, multiple programs can satisfy a given specification:

$$\{X = 1\}\ Y := 2\ \{X = 1 \wedge Y = 2\}$$

$$\{X = 1\}\ Y := X + 1\ \{X = 1 \wedge Y = 2\}$$

$$\{X = 1\}\ Y := 2 * X\ \{X = 1 \wedge Y = 2\}$$

A slightly more complex example (factorial):

Precondition: $\{X > 0 \wedge X = x\}$
1: $F := 1$
2: **while** $X > 0$ **do**
3: $\quad F := F \cdot X$
4: $\quad X := X - 1$
5: **od**
Postcondition: $\{F = x!\}$

Software Verification

8 General Aspects

9 Software Testing

10 Software Specification

11 Software Verification

Floyd-Hoare Logic

- Floyd-Hoare Logic is a set of inference rules that enable us to formally proof partial correctness of a program.
- If S is a statement, we write $\vdash S$ to mean that S has a proof.
- The axioms of Hoare logic will be specified with a notation of the following form:

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}$$

- The conclusion S may be deduced from $\vdash S_1, \ldots, \vdash S_n$, which are the hypotheses of the rule.
- The hypotheses can be theorems of Floyd-Hoare logic or they can be theorems of mathematics.

So far we have discussed the formal specification of software using preconditions and postconditions and we have introduced a simple imperative programming language consisting essentially of variables, expressions and variable assignments, a conditional command, a loop command, and command sequences. The next step is to define inference rules that allow us to make inferences over the commands of this simple programming language. This will give us a formal framework to prove that a program processing input satisfying the precondition will produce a result satisfying the postcondition.

Floyd-Hoare logic is a deductive proof system for Floyd-Hoare triples. It can be used to extract verification conditions (VCs), which are proof obligations or proof subgoals that must be proven so that $\{P\}\ C\ \{Q\}$ is true.

Precondition Strengthening

- If P implies P' and we have shown $\{P'\}\ C\ \{Q\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

$$\frac{\vdash P \rightarrow P', \quad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

- Example: Since $\vdash X = n \rightarrow X + 1 = n + 1$, we can strengthen

$$\vdash \{X + 1 = n + 1\}\ X := X + 1\ \{X = n + 1\}$$

to

$$\vdash \{X = n\}\ X := X + 1\ \{X = n + 1\}.$$

The precondition P is stronger than P' ($P \rightarrow P'$) if the set of states $\{s|s \vdash P\} \subseteq \{s|s \vdash P'\}$.

Precondition strengthening applied to the assignment axiom gives us a triple that feels more intuitive. But keep in mind that $\vdash \{X = n\}\ X := X + 1\ \{X = n + 1\}$ has been derived by combining the assignment axiom with precondition strengthening.

Postcondition Weakening

- If Q' implies Q and we have shown $\{P\}\ C\ \{Q'\}$, then $\{P\}\ C\ \{Q\}$ holds as well:

$$\frac{\vdash \{P\}\ C\ \{Q'\}, \quad \vdash Q' \rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

- Example: Since $X = n + 1 \rightarrow X > n$, we can weaken

$$\vdash \{X = n\}\ X := X + 1\ \{X = n + 1\}$$

to

$$\vdash \{X = n\}\ X := X + 1\ \{X > n\}$$

The postcondition Q is weaker than Q' ($Q' \rightarrow Q$) if the set of states $\{s|s \vdash Q'\} \subseteq \{s|s \vdash Q\}$.

Weakest Precondition

Definition (weakest precondition)

Given a program C and a postcondition Q, the *weakest precondition* $wp(C, Q)$ denotes the largest set of states for which C terminates and the resulting state satisfies Q.

Definition (weakest liberal precondition)

Given a program C and a postcondition Q, the *weakest liberal precondition* $wlp(C, Q)$ denotes the largest set of states for which C leads to a resulting state satisfying Q.

- The "weakest" precondition P means that any other valid precondition implies P.
- The definition of $wp(C, Q)$ is due to Dijkstra (1976) and it requires termination while $wlp(C, Q)$ does not require termination.

In Hoare Logic, we can usually define many valid preconditions. For example, all of the following are valid Hoare triples:

$$\vdash \{X = 1\}\ X := X + 1\ \{X > 0\}$$

$$\vdash \{X > 0\}\ X := X + 1\ \{X > 0\}$$

$$\vdash \{X > -1\}\ X := X + 1\ \{X > 0\}$$

Obviously, the second preconditions is weaker than the first since $X = 1$ implies $X > 0$. With a similar argument, the third precondition is weaker than the second since $X > 0$ implies $X > -1$. How does the precondition $X = 0$ compare to the second and third alternative?

The weakest liberal precondition for $X := X + 1$ and the postcondition $X > 0$ is:

$$wlp(X := X + 1, X > 0) = (X > -1)$$

Since we can assume that the assignment always terminates in this specific case, we have:

$$wp(X := X + 1, X > 0) = wlp(X := X + 1, X > 0) = (X > -1)$$

Strongest Postcondition

Definition (stronges postcondition)

Given a program C and a precondition P, the *strongest postcondition* $sp(C, P)$ has the property that $\vdash \{P\}\ C\ \{sp(C, P)\}$ and for any Q with $\vdash \{P\}\ C\ \{Q\}$, we have $\vdash sp(C, P) \rightarrow Q$.

- The "strongest" postcondition Q means that any other valid postcondition is implied by Q (via postcondition weakening).

Assignment Axiom

- Let $P[E/V]$ (P with E for V) denote the result of substituting the term E for all occurances of the variable V in the statement P.
- An assignment assigns a variable V an expression E:

$$\vdash \{P[E/V]\}\ V := E\ \{P\}$$

- Example:

$$\{X + 1 = n + 1\}\ X := X + 1\ \{X = n + 1\}$$

The assignment axiom kind of works backwards. In the example, we start with P, which is $\{X = n+1\}$. In P, we substitute E, which is $X + 1$, for V, which is X. This gives us $\{X + 1 = n + 1\}$.

Note that the term E is evaluated in a state where the assignment has not yet been carried out. Hence, if a statement P is true after the assignment, then the statement obtained by substituting E for V in P must be true before the assignment.

Two common erroneous intuitions:

1. $\vdash \{P\}\ V := E\ \{P[V/E]\}$

   This has the consequence $\vdash \{X = 0\}\ X := 1\ \{X = 0\}$ since $X = 0[X/1]$ is equal to $X = 0$ (since 1 does not occur in $X = 0$).

2. $\vdash \{P\}\ V := E\ \{P[E/V]\}$

   This has the consequence $\vdash \{X = 0\}\ X := 1\ \{1 = 0\}$ since one would substitute X with 1 in $X = 0$.

Warning: An important assumption here is that expressions have no side effects that modify the program state. The assignment axiom depends on this property. (Many real-world programming languages, however, do allow side effects.) To see why side effects cause problems, consider an expression $(C; E)$ that consists of a command C and an expression E, e.g. $(Y := 1; 2)$. With this, we would get $\vdash \{Y = 0\}\ X := (Y := 1; 2)\ \{Y = 0\}$ (the substitution would not affect Y).

Specification Conjunction and Disjunction

- If we have shown $\{P_1\}\ C\ \{Q_1\}$ and $\{P_2\}\ C\ \{Q_2\}$, then $\{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}$ holds as well:

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

- We get a similar rule for disjunctions:

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

- These rules allows us to prove $\vdash \{P\}\ C\ \{Q_1 \wedge Q_2\}$ by proving both $\vdash \{P\}\ C\ \{Q_1\}$ and $\vdash \{P\}\ C\ \{Q_2\}$.

Skip Command Rule

- Syntax: *SKIP*
- Semantics: Do nothing. The state after execution the command *SKIP* is the same as the state before executing the command *SKIP*.
- Skip Command Rule:

$$\frac{}{\vdash \{P\}\ \mathit{SKIP}\ \{P\}}$$

Sequence Rule

- Syntax: $C_1; \ldots; C_n$
- Semantics: The commands $C_1, \ldots, C_n$ are executed in that order.
- Sequence Rule:

$$\frac{\vdash \{P\}\ C_1\ \{R\}, \quad \vdash \{R\}\ C_2\ \{Q\}}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

The sequence rule can be easily generalized to $n > 2$ commands:

$$\frac{\vdash \{P\}\ C_1\ \{R_1\},\ \vdash \{R_1\}\ C_2\ \{R_2\},\ \ldots,\ \vdash \{R_{n-1}\}\ C_n\ \{Q\}}{\vdash \{P\}\ C_1; C_2; \ldots; C_n\ \{Q\}}$$

Example (swapping two numbers):

Precondition: $\{X = x \wedge Y = y\}$
1: $R := X$
2: $X := Y$
3: $Y := R$
Postcondition: $\{X = y \wedge Y = x\}$

The proof of the correctness of the sequence of assignments is broken down into the following steps:

(i) $\vdash \{X = x \wedge Y = y\}\ R := X\ \{R = x \wedge Y = y\}$ (assignment axiom)

(ii) $\vdash \{R = x \wedge Y = y\}\ X := Y\ \{R = x \wedge X = y\}$ (assignment axiom)

(iii) $\vdash \{R = x \wedge X = y\}\ Y := R\ \{Y = x \wedge X = y\}$ (assignment axiom)

(iv) $\vdash \{X = x \wedge Y = y\}\ R := X; X := Y\ \{R = x \wedge X = y\}$ (sequence rule for (i) and (ii))

(v) $\vdash \{X = x \wedge Y = y\}\ R := X; X := Y; Y := R\ \{Y = x \wedge X = y\}$ (sequence rule for (iv) and (iii))

Conditional Command Rule

- Syntax: *IF S THEN C_1 ELSE C_2 FI*
- Semantics: If the statement S is true in the current state, then C_1 is executed. If S is false, then C_2 is executed.
- Conditional Rule:

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \quad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ \{Q\}}$$

Consider the following specification and program (max):

Precondition: $\{X = x \wedge Y = y\}$
1: **if** $X \geq Y$ **then**
2: $M := X$
3: **else**
4: $M := Y$
5: **fi**
Postcondition: $\{M = max(x, y)\}$

In order to prove the partial correctness of this program, we have to prove the correctness of the two assignments under the statement $X \geq Y$ being either true or false. The application of the assignment axiom gives us the following two statements:

$$\{X = x \wedge Y = y \wedge X \geq Y\}\ M := X\ \{M = x \wedge X = x \wedge Y = y \wedge X \geq Y\}$$

$$\{X = x \wedge Y = y \wedge X < Y\}\ M := Y\ \{M = y \wedge X = x \wedge Y = y \wedge X < Y\}$$

The definition of $max(x, y)$ we are going to use is the following:

$$max(x,y) = \begin{cases} x & x \geq y \\ y & x < y \end{cases}$$

This gives us the following implications:

$$M = x \wedge X = x \wedge Y = y \wedge X \geq Y \rightarrow M = max(x, y)$$

$$M = y \wedge X = x \wedge Y = y \wedge X < Y \rightarrow M = max(x, y)$$

Postcondition weakening gives us:

$$\{X = x \wedge Y = y \wedge X \geq Y\}\ M := X\ \{M = max(x, y)\}$$

$$\{X = x \wedge Y = y \wedge X < Y\}\ M := Y\ \{M = max(x, y)\}$$

Applying the conditional rule, we get:

$$\{X = x \wedge Y = y\}\ \text{IF}\ X \geq Y\ \text{THEN}\ M := X\ \text{ELSE}\ M := Y\ \text{FI}\ \{M = max(x, y)\}$$

While Command Rule

- Syntax: *WHILE S DO C OD*
- Semantics: If the statement S is true in the current state, then C is executed and the WHILE-command is repeated. If S is false, then nothing is done. Thus C is repeatedly executed until the value of S becomes false. If S never becomes false, then the execution of the command never terminates.
- While Rule:

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \mathit{WHILE}\ S\ \mathit{DO}\ C\ \mathit{OD}\ \{P \wedge \neg S\}}$$

P is an invariant of C whenever S holds. Since executing C preserves the truth of P, executing C any numbner of times also preserves the truth of P.

Finding invariants is the key to prove the correctness of while loops. The invariant should

- say what has been done so far together with what remains to be done;
- hold at each iteration of the loop;
- give the desired result when the loop terminates.

Example (factorial):

Precondition: $\{Y = 1 \wedge Z = 0 \wedge X = x \wedge X \geq 0\}$
1: **while** $Z \neq X$ **do**
2: $Z := Z + 1$
3: $Y := Y \cdot X$
4: **od**
Postcondition: $\{Y = x!\}$

We need to find an invariant P such that:

- $\{P \wedge Z \neq X\}\ Z := Z + 1;\ Y := Y \cdot Z\ \{P\}$ (while rule)
- $Y = 1 \wedge Z = 0 \rightarrow P$ (precondition strengthening)
- $P \wedge \neg(Z \neq X) \rightarrow Y = X!$ (postcondition weakening)

The invariant $Y = Z!$ serves the purpose:

- $Y = Z! \wedge Z \neq X \rightarrow Y \cdot (Z + 1) = (Z + 1)!$
  $\{Y \cdot (Z + 1) = (Z + 1)!\}\ Z := Z + 1\ \{Y \cdot Z = Z!\}$ (assignment axiom)
  $\{Y \cdot Z = Z!\}\ Y := Y * Z\ \{Y = Z!\}$ (assignment axiom)
  $\{Y = Z!\}\ Z := Z + 1;\ Y := Y * Z\ \{Y = Z!\}$ (sequence rule)
- $Y = 1 \wedge Z = 0 \rightarrow Y = Z!$ since $0! = 1$
- $Y = Z! \wedge \neg(Z \neq X) \rightarrow Y = X!$ since $\neg(Z \neq X)$ is equivalent to $Z = X$

Arrays

- Let the terms $A\{E_1 \leftarrow E2\}$ denote an array identical to A with the E_1-th component changed to the value E_2.
- With this, the assignment command can be extended to support arrays, i.e., the array assignment is a special case of an ordinary variable assignment.

$$\vdash \{P[A\{E_1 \leftarrow E_2\}/A]\}\ A[E_1] := E_2\ \{P\}$$

- The following axioms are needed to reason about arrays:

$$\vdash A\{E_1 \leftarrow E_2\}[E_1] = E_2$$

$$E_1 \neq E_2\ \rightarrow \vdash A\{E_1 \leftarrow E2\}[E_3] = A[E_3]$$

Example (swapping two array elements):

$\{A[X] = x \wedge A[Y] = y\}$
$R := A[X];\ A[X] := A[Y];\ A[Y] := R$
$\{A[X] = y \wedge A[Y] = x\}$

Working from the postcondition backwards we get:

$\{A\{Y \leftarrow R\}[X] = y \wedge A\{Y \leftarrow R\}[Y] = x\}$
$A[Y] := R$
$\{A[X] = y \wedge A[Y] = x\}$

Applying precondition strengthening, using $A\{Y \leftarrow R\}[Y] = R$, this simplifies to:

$\{A\{Y \leftarrow R\}[X] = y \wedge R = x\}$
$A[Y] := R$
$\{A[X] = y \wedge A[Y] = x\}$

Continuing backwards, we get:

$\{A\{X \leftarrow A[Y]\}\{Y \leftarrow R\}[X] = y \wedge R = x\}$
$A[X] := A[Y]$
$\{A\{Y \leftarrow R\}[X] = y \wedge R = x\}$

Continuing one more step backwards, we get:

$\{A\{X \leftarrow A[Y]\}\{Y \leftarrow A[X]\}[X] = y \wedge A[X] = x\}$
$R := A[X]$
$\{A\{X \leftarrow A[Y]\}\{Y \leftarrow R\}[X] = y \wedge R = x\}$

Applying the sequencing rule, we get:

$\{A\{X \leftarrow A[Y]\}\{Y \leftarrow A[X]\}[X] = y \wedge A[X] = x\}$
$R := A[X];\ A[X] := A[Y];\ A[Y] := R$
$\{A[X] = y \wedge A[Y] = x\}$

Using the array axioms (considering $X = Y$ and $X \neq Y$ separately), we obtain:

$A\{X \leftarrow A[Y]\}\{Y \leftarrow A[X]\}[X] = A[Y]$

This leads us to what we needed to show.

Proof Automation

- Proving even simple programs manually takes a lot of effort
- There is a high risk to make mistakes during the process
- General idea how to automate the proof:
  (i) Let the human expert provide annotations of the specification (e.g., loop invariants) that help with the generation of proof obligations
  (ii) Generate proof obligations automatically (verification conditions)
  (iii) Use automated theorem provers to verify some of the proof obligations
  (iv) Let the human expert prove the remaining proof obligations (or let the human expert provide additional annotations that help the automated theorem prover)
- Step (ii) essentially compiles an annotated program into a conventional mathematical problem.

Consider the following program:

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: **while** $Y \leq R$ **do**
4: $R := R - Y$
5: $Q := Q + 1$
6: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

Annotations

- Annotations are required
  (i) before each command C_i (with $i > 1$) in a sequence $C_1; C_2; \ldots; C_n$, where C_i is not an assignment command and
  (ii) after the keyword *DO* in a *WHILE* command (loop invariant)
- The inserted annotation is expected to be true whenever the execution reaches the point of the annotation.
- For a properly annotated program, it is possible to generate a set of proof goals (verification conditions).
- It can be shown that once all generated verification conditions have been proved, then $\vdash \{P\}\ C\ \{Q\}$.

We add suitable annotations:

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: $\{R = X \wedge Q = 0\}$
4: **while** $Y \leq R$ **do**
5: $\{X = Y \cdot Q + R\}$
6: $R := R - Y$
7: $Q := Q + 1$
8: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

This should (ideally automatically) lead to the following proof obligations (verification conditions):

1. $\top \to (X = X \wedge 0 = 0)$
2. $(R = X \wedge Q = 0) \to (X = Y \cdot Q + R)$
3. $(X = Y \cdot Q + R \wedge \neg(Y \leq R)) \to (X = Y \cdot Q + R \wedge R < Y)$
4. $(X = Y \cdot Q + R \wedge Y \leq R) \to (X = Y \cdot (Q + 1) + (R - Y)$

Generation of Verification Conditions

- Assignment $\{P\}\ V := E\ \{Q\}$:
  Add verification condition $P \rightarrow Q[E/V]$.
- Conditions $\{P\}\ \textit{IF S THEN } C_1 \textit{ ELSE } C_2 \textit{ FI}\ \{Q\}$
  Add verification conditions generated by $\{P \wedge S\}\ C_1\ \{Q\}$ and $\{P \wedge \neg S\}\ C_2\ \{Q\}$
- Sequences of the form $\{P\}\ C_1; \ldots; C_{n-1};\ \{R\}\ C_n\ \{Q\}$
  Add verification conditions generated by $\{P\}\ C_1; \ldots; C_{n-1}\ \{R\}$ and $\{R\}\ C_n\ \{Q\}$
- Sequences of the form $\{P\}\ C_1; \ldots; C_{n-1};\ V := E\ \{Q\}$
  Add verification conditions generated by $\{P\}\ C_1; \ldots; C_{n-1}\ \{Q[E/V]\}$
- While loops $\{P\}\ \textit{WHILE S DO}\ \{R\}\ C\ \textit{OD}\ \{Q\}$
  Add verification conditions $P \rightarrow R$ and $R \wedge \neg S \rightarrow Q$
  Add verificiation conditions generated by $\{R \wedge S\}\ C\ \{R\}$

Starting with the annotated example:

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: $\{R = X \wedge Q = 0\}$
4: **while** $Y \leq R$ **do**
5: $\{X = Y \cdot Q + R\}$
6: $R := R - Y$
7: $Q := Q + 1$
8: **od**
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

According to the second sequence rule, we have to generate VCs for the while loop and the sequence consisting of the initial assignments. The initial assignments reduce to $\top \rightarrow (X = X \wedge 0 = 0)$ as follows:

$$\{\top\}\ R := X;\ Q := 0\ \{R = X \wedge Q = 0\}$$
$$\{\top\}\ R := X\ \{R = X \wedge 0 = 0\}$$
$$\top \rightarrow (X = X \wedge 0 = 0)$$

The while loop rule gives us the following two VCs

$$(R = X \wedge Q = 0) \rightarrow (X = Y \cdot Q + R)$$
$$(X = Y \cdot Q + R \wedge \neg(Y \leq R)) \rightarrow (X = Y \cdot Q + R \wedge R < Y)$$

and the VC generated as follows:

$$\{X = Y \cdot Q + R \wedge Y \leq R\}\ R := R - Y;\ Q := Q + 1\ \{X = Y \cdot Q + R\}$$
$$\{X = Y \cdot Q + R \wedge Y \leq R\}\ R := R - Y;\ \{X = Y \cdot (Q + 1) + R\}$$
$$(X = Y \cdot Q + R \wedge Y \leq R) \rightarrow (X = Y \cdot (Q + 1) + (R - Y))$$

Total Correctness

- We assume that the evaluation of expressions always terminates.
- With this simplifying assumption, only *WHILE* commands can cause loops that potentially do not terminate.
- All rules for the other commands can simply be extended to cover total correctness.
- The assumption that expression evaluation always terminates is often not true. (Consider recursive functions that can go into an endless recursion.)
- We have so far also silently assumed that the evaluation of expressions always yields a proper value, which is not the case for a division by zero.
- Relaxing our assumptions for expressions is possible but complicates matters significantly.

If C does not contain any while commands, then we have the simple rule:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash [P]\ C\ [Q]}$$

Rules for Total Correctness [1/4]

- Assignment axiom

$$\vdash [P[E/V]]\ V := E\ [P]$$

- Precondition strengthening

$$\frac{\vdash P \rightarrow P', \quad \vdash [P']\ C\ [Q]}{\vdash [P]\ C\ [Q]}$$

- Postcondition weakening

$$\frac{\vdash [P]\ C\ [Q'], \quad \vdash Q' \rightarrow Q}{\vdash [P]\ C\ [Q]}$$

Rules for Total Correctness [2/4]

- Specification conjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \wedge P_2]\ C\ [Q_1 \wedge Q_2]}$$

- Specification disjunction

$$\frac{\vdash [P_1]\ C\ [Q_1], \quad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \vee P_2]\ C\ [Q_1 \vee Q_2]}$$

- Skip command rule

$$\overline{[P]\ SKIP\ [P]}$$

Rules for Total Correctness [3/4]

- Sequence rule

$$\frac{\vdash [P]\ C_1\ [R_1],\ \vdash [R_1]\ C_2\ [R_2],\ \ldots,\ \vdash [R_{n-1}]\ C_n\ [Q]}{\vdash [P]\ C_1; C_2; \ldots; C_n\ [Q]}$$

- Conditional rule

$$\frac{\vdash [P \land S]\ C_1\ [Q], \quad \vdash [P \land \neg S]\ C_2\ [Q]}{\vdash [P]\ \mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}\ [Q]}$$

Rules for Total Correctness [4/4]

- While rule

$$\frac{\vdash [P \land S \land E = n]\ C\ [P \land (E < n)], \quad \vdash P \land S \rightarrow E \geq 0}{\vdash [P]\ WHILE\ S\ DO\ C\ OD\ [P \land \neg S]}$$

E is an integer-valued expression
n is an auxiliary variable not occuring in P, C, S, or E

- A prove has to show that a non-negative integer, called a *variant*, decreases on each iteration of the loop command C.

We show that the while loop in the following program terminates.

Precondition: $\{\top\}$
1: $R := X$
2: $Q := 0$
3: **while** $Y \leq R$ **do**
4: $R := R - Y$
5: $Q := Q + 1$
6: **od**
Postcondition: $\{X = Y \cdot Q + R \land R < Y\}$

We apply the while rule with

$$P = Y > 0$$
$$S = Y \leq R$$
$$E = R$$

and we have to show the following to be true:

1. $[P \land S \land E = n]\ R := R - Y; Q := Q + 1\ [P \land (E < n)]$

   This follows from the following derivation:

   $$[P \land S \land E = n]\ R := R - Y; Q := Q + 1\ [P \land (E < n)]$$
   $$[Y > 0 \land Y \leq R \land R = n]\ R := R - Y; Q := Q + 1\ [Y > 0 \land (R < n)]$$
   $$Y > 0 \land Y \leq R \land R = n \rightarrow Y > 0 \land (R < n)[Q + 1/Q][R - Y/R]$$
   $$Y > 0 \land Y \leq R \land R = n \rightarrow Y > 0 \land ((R - Y) < n)$$

2. $P \land S \rightarrow E \geq 0$

   This follows from:

   $$P \land S \rightarrow E \geq 0$$
   $$Y > 0 \land Y \leq R \rightarrow R > 0$$

Generation of Termination Verification Conditions

- The rules for the generation of termination verificiation conditions follow directly from the rules for the generation of partial correctness verificiation conditions, except for the while command.
- To handle the while command, we need an additional annotation (in square brackets) that provides the variant expression.
- For while loops of the form $\{P\}$ *WHILE S DO* $\{R\}$ $[E]$ *C OD* $\{Q\}$ add the verification conditions

$$P \rightarrow R$$
$$R \wedge \neg S \rightarrow Q$$
$$R \wedge S \rightarrow E \geq 0$$

and add verificiation conditions generated by $\{R \wedge S \wedge (E = n)\}$ C $\{R \wedge (E < n)\}$

Annotated example including the variant annotation for termination verification rule generation:

Precondition: $\{\top\}$

1: $R := X$
2: $Q := 0$
3: $\{R = X \wedge Q = 0\}$
4: **while** $Y \leq R$ **do**
5: $\{X = Y \cdot Q + R\}$
6: $[R]$
7: $R := R - Y$
8: $Q := Q + 1$
9: **od**

Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$

The while loop rule gives use the following termination VCs

$$(R = X \wedge Q = 0) \rightarrow (X = Y \cdot Q + R)$$
$$(X = Y \cdot Q + R \wedge \neg(Y \leq R)) \rightarrow (X = Y \cdot Q + R \wedge R < Y)$$
$$(X = Y \cdot Q + R \wedge (Y \leq R)) \rightarrow R \geq 0$$

and the VC generated as follows:

$$\{X = Y \cdot Q + R \wedge Y \leq R \wedge R = n\}\ R := R - Y;\ Q := Q + 1\ \{X = Y \cdot Q + R \wedge R < n\}$$
$$\{X = Y \cdot Q + R \wedge Y \leq R \wedge R = n\}\ R := R - Y;\ \{X = Y \cdot (Q + 1) + R \wedge R < n\}$$
$$(X = Y \cdot Q + R \wedge Y \leq R \wedge R = n) \rightarrow (X = Y \cdot (Q + 1) + (R - Y) \wedge (R - Y) < n)$$

The last VC is not true in general and hence the algorithm does not always terminate:

$Y = 0:$

$$((X = R \wedge 0 \leq R \wedge R = n) \rightarrow (X = R \wedge R < n)) \rightarrow \bot$$

$Y < 0:$

$$((X = Y \cdot Q + R \wedge Y \leq R \wedge R = n) \rightarrow (X = Y \cdot (Q + 1) + (R - Y) \wedge (R - Y) < n)) \rightarrow \bot$$

Termination and Correctness

- Partial correctness and termination implies total correctness:

$$\frac{\vdash \{P\}\ C\ \{Q\}, \quad \vdash [P]\ C\ [\mathbf{T}]}{\vdash [P]\ C\ [Q]}$$

- Total correctness implies partial correctness and termination:

$$\frac{\vdash [P]\ C\ [Q]}{\vdash \{P\}\ C\ \{Q\}, \quad \vdash [P]\ C\ [\mathbf{T}]}$$

Part III
Concurrency and Distributed Algorithms

Most larger programs these days are concurrent and the dependability of complex software systems often depends on how concurrency requirements have been addressed. It is thus useful to look at basic ideas to model concurrency.

Some programming languages address concurrency as part of their language design. The Erlang language is build on the actor model, while the more recent Go language is build on the communicating sequential processes model (although not purely).

Note that concurrency should not be confused with parallelism. Concurrency deals with tasks that can start, run, and complete in overlapping time periods. Concurrency does not necessarily mean the tasks will ever be running at the same time. Parallelism is when tasks are simultaneously executed, e.g., on a multicore processor. As Rob Pike once said, "concurrency is about dealing with lots of things at once while parallelism is about doing lots of things at once".

The discussion of basic concepts to model distributed algorithms follows Gerald Tel's book on distributed algorithms [29]. We focus, however, on the parts that provide an basic understanding of the aspects of building distributed systems that are fault tolerant.

The discussion of communicating sequential processes follows Tony Hoare's book on communicating sequential processes [14].

Modeling Concurrent Systems

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Actor Model (Hewitt 1973)

- An *actor* is a computational entity that, in response to a message it receives, can concurrently:
  - send a finite number of messages to other actors;
  - create a finite number of new actors;
  - designate the behavior to be used for the next message it receives.
- There is no assumed order on the actions and they could be carried out in parallel.
- Everything is an actor. An actor can only communicate with actors whose addresses it has.
- Actors are concurrent, interaction only through direct asynchronous message passing.

Communicating Sequential Processes (Hoare 1978)

- Communicating Sequential Processes (CSP) were proposed as a foundation for a concurrent programming language and the ideas were later formalized into a calculus belonging to the family of process calculi, mathematical formalisms for describing and analyzing properties of concurrent computation.
- CSP is based on events and processes and a message passing idea using channels.
- CSP processes are anonymous, actors have identified names.
- CSP message-passing fundamentally involves a rendezvous between the processes involved in sending and receiving the message.
- CSP uses explicit channels for message passing, whereas actor systems transmit messages to named destination actors.

The original CSP paper [14] appeared in 1978 and was very close to programming language concepts but the CSP idea has evolved significantly since then towards a mathematical model. A very good introduction is the CSP book [15], which you can find online (but note the usage restrictions).

The CSP model has influenced the design of programming languages, most recently perhaps that design of the Go programming language. The Go programming languages has lightweight threads (called goroutines in Go lingo) that are mapped dynamically to a pool of system-level threads, where the pool of system-level threads is typically a function of the number of available cores. Instead of using shared memory and primitives like semaphores or locks and condition variables, the designers of Go prefer a message passing paradigm where data is exchanged between goroutines using channels.

Logical Clocks (Lamport 1978)

- Analyzing distributed systems requires to understand causality.
- It is important to know what happend before a certain event that can have influenced the event.
- Regular time does not provide a good way to express an order of events in a distributed system (clock synchronization issues)
- Lamport proposed logical clocks that can express the *happened-before* relation on the set of events.
- While Lamport clocks are able to express the *happened-before* relationship, they are insufficient to express causality or concurrency.

The original paper introducing logical clocks is [17]. A good summary of basic logical clocks and extensions such as vector clocks and matrix clocks can be found in [25].

π Calculus (Milner 1992)

- The π-calculus belongs to the family of process calculi, mathematical formalisms for describing and analyzing properties of concurrent computation.
- The aim of the π-calculus is to be able to describe concurrent computations whose configuration may change during the computation
- The π-calculus is general (turing complete).
- The π-calculus has been extended with cryptographic primitives to form the spi-calculus, which has been used to analyze cryptographic protocols.

The π-calculus is described in [20] and the Spi-calculus for cryptographic protocols in [1].

Model of Distributed Algorithms

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Transition System

Definition (transition system)

A transition system is a triple $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ where $\mathcal{C}$ is a set of configurations, $\rightarrow$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I}$ is a subset of $\mathcal{C}$ of initial configurations.

Definition (execution)

Let $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ be a transition system. An execution of S is a maximal sequence $E = (\gamma_0, \gamma_1, \ldots)$, where $\gamma_0 \in \mathcal{I}$ and $\gamma_i \rightarrow \gamma_{i+1}$ for all $i \geq 0$.

- A transition relation is a subset of $\mathcal{C} \times \mathcal{C}$.
- The notation $\gamma \rightarrow \delta$ is used for $(\gamma, \delta) \in \rightarrow$.

Transition System

Definition (reachability)

Configuration δ is reachable from γ, notation $\gamma \leadsto \delta$, if there exists a sequence $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_k = \delta$ with $\gamma_i \rightarrow \gamma_{i+1}$ for all $0 \leq i < k$.

- A terminal configuration is a configuration γ for which there is no δ such that $\gamma \rightarrow \delta$
- A sequence $E(\gamma_0, \gamma_1, \ldots)$ with $\gamma_i \rightarrow \gamma_{i+1}$ for all i is maximal if it is either infinite or ends in a terminal configuration
- Configuration δ is said to be reachable if it is reachable from an initial configuration.

Local Algorithm

Definition (local algorithm)

The local algorithm of a process is a quintuple $(Z, I, \vdash^i, \vdash^s, \vdash^r)$, where Z is a set of states, I is a subset of Z of initial states, $\vdash^i$ is a relation on $Z \times Z$, and $\vdash^s$ and $\vdash^r$ are relations on $Z \times \mathcal{M} \times Z$. The binary relation $\vdash$ on Z is defined by

$$c \vdash d \iff (c, d) \in \vdash^i \vee \exists m \in \mathcal{M} : (c, m, d) \in (\vdash^s \cup \vdash^r).$$

- Let $\mathcal{M}$ be a set of possible messages. We denote the collection of multisets with elements from $\mathcal{M}$ with $\mathbf{M}(\mathcal{M})$.
- The relations $\vdash^i$, $\vdash^s$, and $\vdash^r$ correspond to state transitions related with internal, send, and receive events.

A local algorithm is simply a sequence of local state transformations, send events and receive events. This is a rather abstract view but our goal is to focus on the interaction between local algorithms.

Distributed Algorithm

Definition (distributed algorithm)

A distributed algorithm for a collection $\mathbb{P} = \{p_1, \ldots, p_N\}$ of processes is a collection of local algorithms, one for each process in $\mathbb{P}$.

- A configuration of a transition system consists of the state of each process and the collection of messages in transit
- The transitions are the events of the processes, which do not only affect the state of the process, but can also affect (and be affected by) the collection of messages
- The initial configurations are the configurations where each process is in an initial state and the message collection is empty

We are constructing a distributed algorithm out of local algorithms. Note that we make some simplifying assumptions here. One of them is that we assume the set of processes involved in a distributed algorithms to be constant.

We use the term process in a generic sense. A process in our context does not have to be an operating system level process. It can as well be a thread or a corouting in a concurrent program (but we assume that all processes only exchange data via send and receive primitives, i.e., there is no shared memory).

Induced Async. Transition System

Definition (Induced Async. Transition System)

The transition system $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ is induced under asynchronous communication by a distributed algorithm for processes $p_1, \ldots, p_N$, where the local algorithm for process p_i is $(Z_{p_i}, I_{p_i}, \vdash^i_{p_i}, \vdash^s_{p_i}, \vdash^r_{p_i})$, is given by

(1) $\mathcal{C} = \{(c_{p_1}, \ldots, c_{p_N}, M) : (\forall p \in \mathbb{P} : c_p \in Z_p) \wedge M \in \mathbf{M}(M)\}$

(2) $\rightarrow$ (see next slide)

(3) $\mathcal{I} = \{(c_{p_1}, \ldots, c_{p_N}, M) : (\forall p \in \mathbb{P} : c_p \in I_p) \wedge M = \emptyset\}$

An asynchronous transition system can store messages that are in transit. A sending process does not have to synchronize with a receiving process in order to send a message. Sending and receiving of messages is asynchronous, which implies that messages in transit can be buffered somewhere.

Asynchronous systems are quite common when we think of communication protocols and communication middleware such as message queues.

Induced Async. Transition System

Definition (Induced Async. Transition System (cont.))

(2) $\rightarrow = (\bigcup_{p \in \mathbb{P}} \rightarrow_p)$, where the $\rightarrow_p$ are the transitions corresponding to the state changes of process p; $\rightarrow_{p_i}$ is the set of pairs

$$(c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_N}, M_1), (c_{p_1}, \ldots, c'_{p_i}, \ldots, c_{p_N}, M_2)$$

for which one of the following three conditions holds:

- $(c_{p_i}, c'_{p_i}) \in \vdash^i_{p_i}$ and $M_1 = M_2$
- for some $m \in \mathcal{M}$, $(c_{p_i}, m, c'_{p_i}) \in \vdash^s_{p_i}$ and $M_2 = M_1 \cup \{m\}$
- for some $m \in \mathcal{M}$, $(c_{p_i}, m, c'_{p_i}) \in \vdash^r_{p_i}$ and $M_1 = M_2 \cup \{m\}$

Induced Sync. Transition System

Definition (Induced Sync. Transition System)

The transition system $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ is induced under synchronous communication by a distributed algorithm for processes $p_1, \ldots, p_N$, where the local algorithm for process p_i is $(Z_{p_i}, I_{p_i}, \vdash^i_{p_i}, \vdash^s_{p_i}, \vdash^r_{p_i})$, is given by

(1) $\mathcal{C} = \{(c_{p_1}, \ldots, c_{p_N}) : (\forall p \in \mathbb{P} : c_p \in Z_p)\}$

(2) $\rightarrow$ (see next slide)

(3) $\mathcal{I} = \{(c_{p_1}, \ldots, c_{p_N}) : (\forall p \in \mathbb{P} : c_p \in I_p)\}$

A synchronous transition system does not store messages in transit. A sending process has to block until a receiving process is willing to receive a message.Sending and receiving of messages is synchronous.

Synchronous systems are less common in practice but important to study. Algorithms for a given distributed computing problem can look very different for synchronous and asynchronous transition systems.

Induced Sync. Transition System

Definition (Induced Sync. Transition System (cont.))

(2) $\rightarrow = (\bigcup_{p \in \mathbb{P}} \rightarrow_p) \cup (\bigcup_{p,q \in \mathbb{P}: p \neq q} \rightarrow_{pq})$, where

- $\rightarrow_{p_i}$ is the set of pairs

$$(c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_N}), (c_{p_1}, \ldots, c'_{p_i}, \ldots, c_{p_N})$$

  for which $(c_{p_i}, c'_{p_i}) \in \vdash^i_{p_i}$
- $\rightarrow_{p_i p_j}$ is the set of pairs

$$(\ldots, c_{p_i}, \ldots, c_{p_j}, \ldots)(\ldots, c'_{p_i}, \ldots, c'_{p_j}, \ldots)$$

  for which there is a message $m \in M$ such that

$$(c_{p_i}, m, c'_{p_i}) \in \vdash^s_{p_i} \text{ and } (c_{p_j}, m, c'_{p_j}) \in \vdash^r_{p_j}$$

Events, Causality, Logical Clocks

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Events and Causal Order

- A transition a is said to occur earlier than transition b if a occures in the sequence of transitions before b
- An execution $E = (\gamma_0, \gamma_1, \ldots)$ can be associated with a sequence of events $\bar{E} = (e_0, e_1, \ldots)$, where e_i is the event by which the configuration changes from γ_i to γ_{i+1}
- Events of a distributed execution can sometimes be interchanged without affecting the later configurations of the execution
- The notion of time as a total order on the events is not suitable and instead the notion of causal dependence is introduced

Dependence of Events

Theorem

Let γ be a configuration of a distributed system (with asynchronous message passing) and let e_p and e_q be events of different processes p and q, both applicable in γ. Then e_p is applicable in $e_q(\gamma)$, e_q is applicable in $e_p(\gamma)$, and $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$.

- Let e_p and e_q be two events that occur consecutively in an execution. The premise of the theorem applies to these events except in the following two cases:
  a) $p = q$ or
  b) e_p is a send event, and e_q is the corresponding receive event
- The fact that a particular pair of events cannot be exchanged is expressed by saying that there is a *causal relation* between these two events

The theorem requires that p and q are different. If e_p is a send event and e_q is a receive event, then both event cannot be applicable in γ. (There is no way to receive a message that has not yet been sent.)

Causal Order

Definition (causal order)

Let E be an execution. The relation $\prec$, called the causal order, on the events of the execution is the smallest relation that satisfies the following requirements:

(1) If e and f are different events of the same process and e occurs before f, then $e \prec f$.

(2) If s is a send event and r the corresponding receive event, then $s \prec r$.

(3) The relation $\prec$ is transitive.

- Let $a \preceq b$ denote $(a \prec b \vee a = b)$; the relation $\preceq$ is a partial order
- There may be events a and b for which neither $a \prec b$ nor $b \prec a$ holds; such events are said to be concurrent, notation $a \mid b$

Understanding the causal order between events in a distributed system is crucial for reasoning about the system. Lamport introduced the causal order relation in his famous paper as the “happened-before” relation between events [17].

Computations

- The events of an execution can be reordered in any order consistent with the causal order, without affecting the result of the execution
- Such a reordering of the events gives rise to a different sequence of configurations, but this execution will be regarded as equivalent to the original execution
- Let $E = (\gamma_0, \gamma_1, \ldots)$ be an execution with an associated sequence of events $\bar{E} = (e_0, e_1, \ldots)$, and assume f is a permutation of $\bar{E}$
- The permutation $(f_0, f_1, \ldots)$ of the events of E is consistent with the causal order if $f_i \preceq f_j$ implies $i \leq j$, i.e., if no event is preceded in the sequence by an event it causally precedes

Equivalent Executions

Theorem

Let $f = (f_0, f_1, \ldots)$ be a permutation of the events of E that is consistent with the causal order of E. Then f defines a unique execution F starting in the initial configuration of E. F has as many events as E, and if E is finite, the last configuration of F is the same as the last configuration of E.

- If the conditions of this theorem apply, we say that E and F are *equivalent* executions, denoted as $E \sim F$
- A global observer, who has access to the actual sequence of events, may distinguish between two equivalent executions
- The processes, however, cannot distinguish between two equivalent executions

Computation

Definition (computation)

A computation of a distributed algorithm is an equivalence class under $\sim$ of executions of the algorithm.

- It makes no sense to speak about the configurations of a computation, because different executions of the computation may not have the same configurations
- It does make sense to speak about the collection of events of a computation, because all executions of the computation consist of the same set of events
- The causal order of the events is defined for a computation

It may be worth to stress that we often do not know the execution of a distributed algorithm. Debugging and monitoring tools may give us an observation of the execution of a distributed algorithms but (i) such tools may change the execution and (ii) different tools may give us different observations of the same execution. In other words, a “perfect” observation of the execution of an algorithm is often not feasible. Furthermore, repeated executions of the same distributed algorithm (with the exact same inputs) may still lead to different executions.

Logical Clocks

Definition (clock)

A clock is a function Θ from the set of events $\bar{E}$ to an ordered set $(X, <)$ such that for $a, b \in \bar{E}$

$$a \prec b \Rightarrow \Theta(a) < \Theta(b).$$

Definition (lamport clock)

A Lamport clock is a clock function Θ_L which assigns to every event a the length k of the longest sequence $(e_1, \ldots, e_k)$ of events satisfying $e_1 \prec e_2 \prec \ldots \prec e_k = a$.

- A clock function Θ expresses causal order, but does not necessarily express concurrency

Lamport Clocks

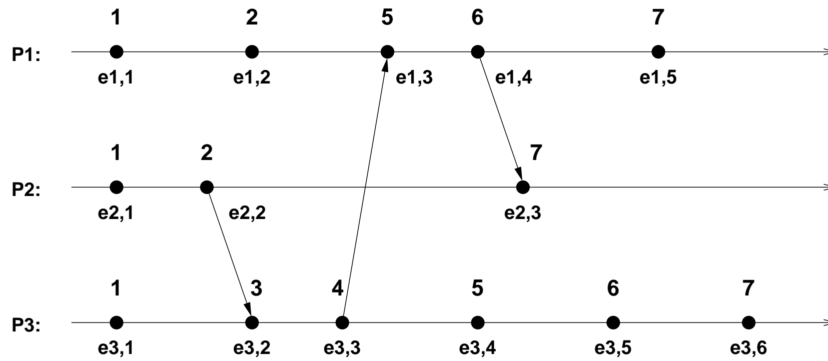
- The value of Θ_L can be computed as follows:
  - $\Theta_L(a)$ is 0 if a is the first event in a process
  - If a is an internal event or send event, and a' the previous event in the same process, then
    $$\Theta_L(a) = \Theta_L(a') + 1$$
  - If a is a receive event, a' the previous event in the same process, and b the send event corresponding to a, then
    $$\Theta_L(a) = \max(\Theta_L(a'), \Theta_L(b)) + 1$$
- The per process clock value may be combined with a process identifier to obtain a globally unique value

A nice property of Lamport clocks is that they are very simple to implement. Every process only needs to maintain a counter and messages exchanged between processes need to carry a counter value. Both are can be implemented with very limited additional resources.

The Ricard-Agrawala algorithm [28] uses Lamport clocks to achieve mutual exclusion of multiple distributed processes that do not share memory.

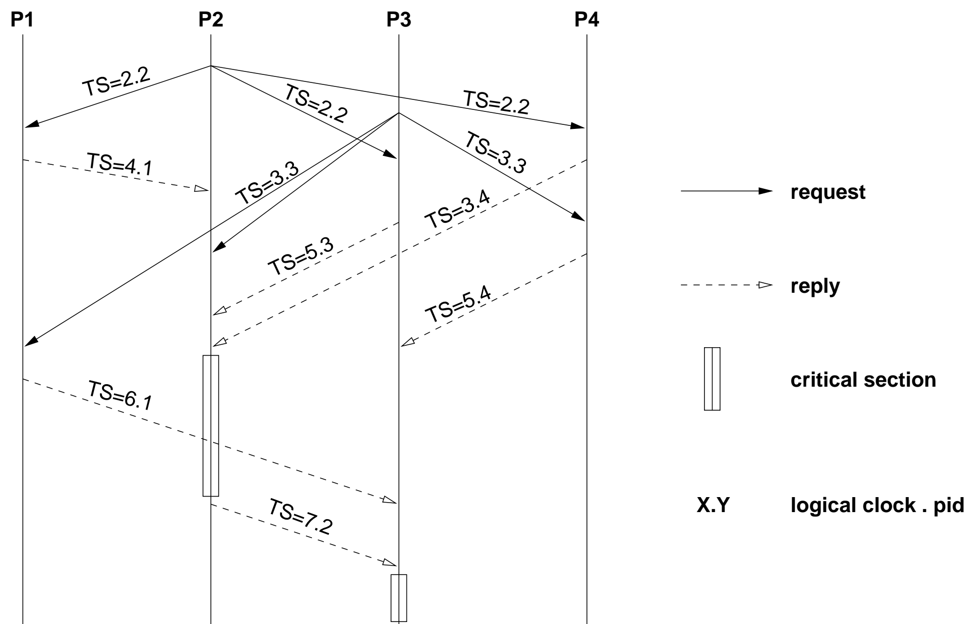
- When process p_i wants to enter its critical section, it generates a new timestamp, t, and sends the message request $< p_i, t_i >$ to all other processes in the system.
- When process p_j receives a request message, it may reply immediately or it may defer sending a reply back. The decision whether process p_j replies immediately to a request $< p_i, t_i >$ message or defers its reply is based on three factors:
  1. If p_j is in its critical section, then it defers its reply to p_i.
  2. If p_j did not request itself to enter its critical section, then it sends a reply immediately to p_i.
  3. If p_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp t_j with the timestamp t_i:
     - If its own request timestamp t_j is greater than t_i, then it sends a reply immediately to p_i (p_i asked first).
     - Otherwise, the reply is deferred until p_j leaves the requested critical section.
- When process p_i receives a reply message from all other processes in the system, it can enter its critical section.
- After exiting its critical section, the process p_i sends reply messages to all its deferred requests.

Lamport Clock Example



Note that Lamport Clock values always identify the longest sequence $(e_1, \ldots, e_k)$ of events satisfying $e_1 \prec e_2 \prec \ldots \prec e_k$. Events can be moved on the time line as long as the causal order is not changed.

Example execution of the Ricard-Agrawala algorithm:



Vector Clocks

Definition (vector clocks)

A vector clock for a set of N processes is a clock function Θ_V which is defined by $\Theta_V(a) = (a_1, \ldots, a_N)$, where a_i is the number of events e in process p_i for which $e \prec a$.

- Vectors are naturally ordered by the vector order:

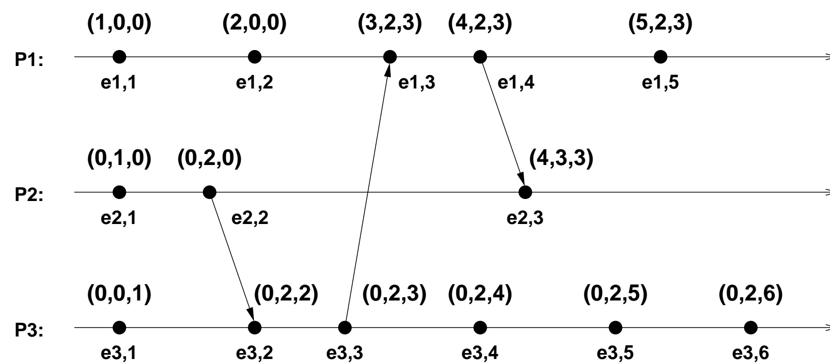
$$(a_1, \ldots a_n) \leq (b_1, \ldots b_n) \iff \forall i \, (1 \leq i \leq b) : a_i \leq b_i$$
$$(a_1, \ldots a_n) < (b_1, \ldots b_n) \iff (a_1, \ldots a_n) \leq (b_1, \ldots b_n) \wedge \exists i : a_i < b_i$$

- Vector clocks can express concurrency since concurrent events are labelled with incomparable clock values:

$$a \prec b \iff \Theta_V(a) < \Theta_V(b)$$
$$a \parallel b \iff \neg(\Theta_V(a) < \Theta_V(b)) \wedge \neg(\Theta_V(b) < \Theta_V(a))$$

Vector clocks require more space than Lamport clocks, which is in particular a concern for messages carrying vector clock values. However, compression techniques can significantly reduce message overhead since often only a small number of vector elements change.

Vector Clock Example



Note that the sum of the components of the Vector Clock values identifies the number of all events causally preceeding the current event, including the current event.

There are even more complex clocks such as matrix clocks. Further details can be found in a nice overview paper written by M. Raynal [25].

Stable Properties and Snapshots

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Properties of Computations

- It is often required to analyze certain *properties* of a computation.
- An important class of properties are so called *stable properties*. A property P of configurations is stable if
$$P(\gamma) \wedge \gamma \rightsquigarrow \delta \Rightarrow P(\delta).$$
- If a computation ever reaches a configuration γ for which P holds true, P remains true in every configuration δ from then on.
- Examples of stable properties: termination, deadlock, loss of tokens, non-reachable objects in dynamic memory structures, ...
- Stable properties can be analyzed off-line by taking a snapshot of a computation.

Stable Property Detection Algorithm

```
1: procedure STABLE-PROPERTY-DETECTION(P)
2:     repeat
3:         γ ← take_global_snapshot()
4:         detected ← P(γ)
5:         if ¬detected then
6:             suspend_some_time()
7:         end if
8:     until detected
9: end procedure
```


One way to detect a stable property P is to take global snapshots and to evaluate P on them until P becomes true. While this is a generic solution for arbitrary stable properties P, it is possible to find more efficient solutions for specific properties. (Remember that recording local snapshots is usually an expensive operation.)

Snapshots Preliminaries

- Let C be a computation of a distributed system consisting of a set of $\mathbb{P}$ processes. The set of events of the computation C is denoted Ev.
- The local computation of process p consists of a sequence $c_p^{(0)}, c_p^{(1)}, \ldots$ of process states, where $c_p^{(0)}$ is an initial state of process p.
- The transition from state $c_p^{(i-1)}$ to $c_p^{(i)}$ is marked by the occurrence of an event $e_p^{(i)}$.
- It follows that $Ev = \bigcup_{p \in \mathbb{P}} \{e_p^{(1)}, e_p^{(2)}, \ldots\}$.

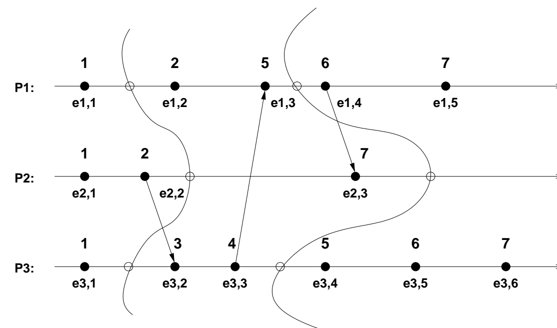
Snapshot Approach (1/2)

- Goal: construct a system configuration composed from local states (snapshot states).
- The local state c_p^* of a process p is called its local *snapshot state*.
- If the snapshot state is $c_p^{(i)}$, i.e., p takes its snapshot between $e_p^{(i)}$ and $e_p^{(i+1)}$, the events $e_p^{(j)}$ with $j \leq i$ are called *preshot events* of p and the event with $j > i$ are called *postshot events* of p
- A (global) snapshot consists of a snapshot state c_p^* for each process p; we write $S^* = (c_{p_1}^*, \ldots, c_{p_N}^*)$
- In time diagrams, local snapshots are depicted by open circles.

Snapshot Approach (2/2)

- If a channel from p to q exists, then the state $c_p^{(i)}$ of p includes a list $sent_{pq}^{(i)}$ of all messages that p has sent to q in the events $e_p^{(1)}$ through $e_p^{(i)}$.
- The state $c_q^{(i)}$ of q includes a list $rcvd_{pq}^{(i)}$ of all messages that q has received from p in the events $e_p^{(1)}$ through $e_p^{(i)}$.
- The state of channel pq is defined to be the set of messages sent by p (according to c_p^*) but not received by q (according to c_q^*); that is $sent_{pq}^* \setminus rcvd_{pq}^*$.
- The simplification ensures that the channel state is recorded in the local snapshots. Note that this assumption can be lifted later on to avoid the storage of all messages.

Anomalies



- Anomalies exist if $rcvd^*_{pq}$ is not a subset of $sent^*_{pq}$
- Anomalies occur if a post-shot message in the snapshot of one process is a pre-shot message in the snapshot of another process.

Feasibile Snapshot and Cuts

Definition (feasible snapshot)

Snapshot S^* is feasible if for each two (neighbor) processes p and q, $rcvd^*_{pq} \subseteq sent^*_{pq}$.

Definition (cut)

A cut of Ev is a set $L \subseteq Ev$ such that

$$e \in L \wedge e' \prec_p e \Rightarrow e' \in L.$$

Cut L_2 is said to be later than L_1 if $L_1 \subseteq L_2$.

Note that a cut only requires that

$$e' \prec_p e \Rightarrow e' \in L$$

and this constraint does not apply to events in other processes.

Consistent Cuts and Meaningful Snapshot

Definition (consistent cut)

A consistent cut of Ev is a set $L \subseteq Ev$ such that

$$e \in L \land e' \prec e \Rightarrow e' \in L.$$

Definition

Snapshot S^* is meaningful in computation C if there exists an execution $E \in C$ such that S^* is a configuration of E.

A consistent cut requires that

$$e' \prec e \Rightarrow e' \in L$$

and this applies to events e' also in other processes.

Feasible vs. Meaningful Snapshots vs. Consistent Cuts

Theorem

Let S^* be a snapshot and L the cut implied by S^*. The following statements are equivalent.

(1) *S^* is feasible.*

(2) *L is a consistent cut.*

(3) *S^* is meaningful.*

- Note that feasibility is a local property between neighbors, while meaningfulness is a global property of the snapshot.

The proof of the theorem shows that (1) implies (2), (2) implies (3), and (3) implies (1). See Gerard Tel [29] for the details.

Chandy-Lamport Algorithm

```
1: procedure INITIATE
2:     if ¬taken_p then
3:         record_local_state()
4:         taken_p ← true
5:         for ∀q ∈ Neigh_p do
6:             send(q, marker)
7:         end for
8:     end if
9: end procedure
```

```
1: procedure MARKER-ARRIVED
2:     recv(q, marker)
3:     if ¬taken_p then
4:         record_local_state()
5:         taken_p ← true
6:         for ∀q ∈ Neigh_p do
7:             send(q, marker)
8:         end for
9:     end if
10: end procedure
```


Every process p has a boolean variable $taken_p$, which is initially set to false. The variable $Neigh_p$ holds the set of neighbors of p.

The Chandy-Lamport snapshot algorithm requires the exchange of $N(N-1)$ messages with N processes. For the original publication, see [6].

Chandy-Lamport Properties

- The channels are assumed to be first in – first out (FIFO), i.e., they do not reorder messages.
- Processes inform each other about snapshot construction by sending special marker messages.
- The algorithm must be initiated by at least one process, but it works correctly if initiated by an arbitrary non-empty set of processes.
- The algorithm of Chandy-Lamport computes a meaningful snapshot within finite time after its initialization by at least one process.

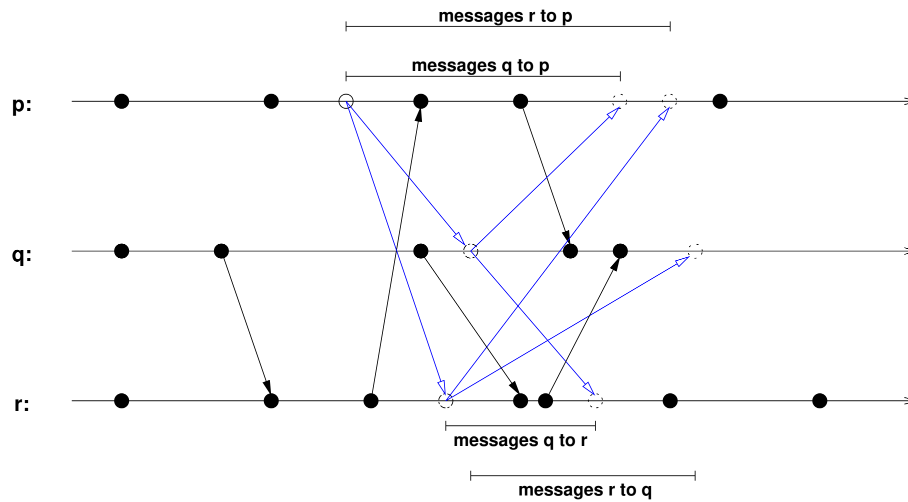
Construction of the Channel State

Lemma

In a feasible snapshot, $send^*_{pq} \setminus rcvd^*_{pq}$ equals the set of messages sent by p in a preshot event and received by q in a postshot event if the channels have FIFO property.

- Chandy-Lamport Algorithm:
  - All preshot messages from p to q are received before the marker message sent from p to q.
  - Moreover, only preshot messages are received before the marker.
  - The state of the channel pq is the collection of messages received by q after recording its state but before the receipt of p's marker message.

Chandy-Lamport Snapshot Example



We have three processes p, q, and r. Process p initiates the snapshot and sends a marker to q and r after taking a local snapshot. When the processes q and r receive the marker, they take their local snapshots and they inform the other processes by sending the marker again.

After taking a local snapshot, the processes have to record messages received from processes that did not initiate their local snapshot until the second markers have arrived.

Discussion points:

- What happens if p's first marker to r arrives after q's marker has arrived at r?
- Which messages are recorded by which process?
- With three processes, we have six channels ($p \to q$, $p \to r$, $q \to p$, $q \to r$, $r \to p$, $r \to q$). Why is it sufficient to record messages only from four channels?
- What happens if p and r initiate the algorithm concurrently?

Fault Tolerance and Broadcasts

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

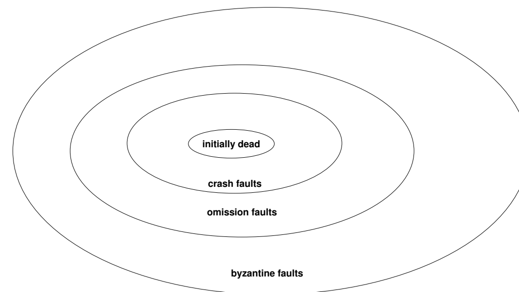
17 Communicating Sequential Processes

Fault Models

- *Initially dead:* The fault that causes a component to not participate during the lifetime of the system.
- *Crash fault:* The fault causes the component to halt or to lose its internal state.
- *Omission fault:* A fault that causes a component to not respond to some input.
- *Timing fault:* A fault that causes a component to respond either too early or too late.
- *Byzantine fault:* An arbitrary fault which causes the component to behave in a totally arbitrary manner during failure.

It is important to be explicit about the fault model. A distributed algorithm may be robust against some faults but not robust against some other faults.

Hierarchy of Fault Models



⟹ Incorrect computation faults are a subset of Byzantine faults where a component does not have any timing fault, but simply produces an incorrect output in response to the given input.

Note that an omission fault is a special case of a timing fault.

Benign vs. Malign Failures

- Initially dead processes and crashes are called *benign* failure types.
- Byzantine failures, which are not benign failures, are called *malign* failure types.
- For several distributed problems, it turns out that a collection of N processes can tolerate $< \frac{N}{2}$ benign failures.
- For several distributed problems in asynchronous distributed systems, it turns out that a collection of N processes can tolerate $< \frac{N}{3}$ malign failures.
- For several distributed problems in a synchronous system, a higher level of robustness can be achieved, especially if messages can be signed.

$\Longrightarrow$ Note that synchronous systems allow for timing errors which do not exist in asynchronous systems.

It can be shown that the decision problem cannot be solved in an asynchronous distributed system if there is at least one crash process.

Approaches to Fault-Tolerance

- Robust Algorithms
  - Correct processes should continue behaving correctly in spite of failures
  - Tolerate failures by using replication and voting
  - Never wait for all processes because processes can fail
- Stabilizing Algorithms (sometimes self-stabilizing Algorithms)
  - Correct processes might be affected by failures, but will eventually become correct
  - The system can start in any state (possibly faulty), but should eventually resume correct behavior

There has been quite some interest in stabilizing algorithms. The firefly algorithm (usually shown during orientation week) is an example of a self-stabilizing algorithm.

For robust algorithms, a fairly general problem is the distributed decision problem.

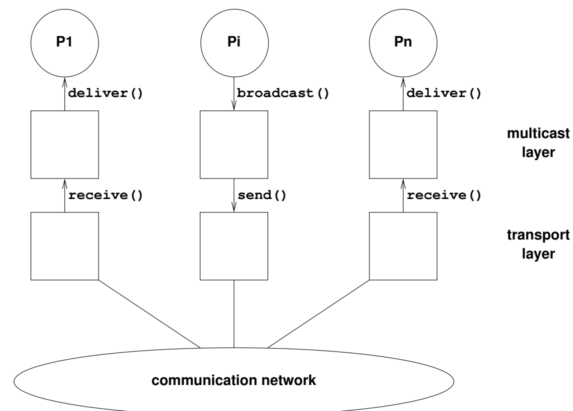
Robust Decision Algorithms

- Robust algorithms typically try to solve some decision problem where each correct process irreversibly decides.
- Three requirements on distributed decision problems:
  1. Termination: All correct processes eventually decide
  2. Consistency: Constraint on different processes decisions
     - Consensus problem: All decisions should be equal
     - Election: All decisions except one should be the same
  3. Non-triviality: Fixed trivial outputs (e.g., always decide "yes") are excluded; processes need to communicate to be able to solve the problem
- Example: All processes in a distributed databases must agree whether to commit or abort a transaction.

Reliable Broadcasts

- Reliable Broadcast:
  - All correct processes deliver the same set of messages
  - The set only contains messages from correct processes
- Atomic Broadcast (reliable)
  - A reliable broadcast where it is guaranteed that every process receives its messages in the same order as all the other processes
- Given a reliable atomic broadcast, we can implement a consensus algorithm
  - Let every node broadcast either 0 or 1
  - Decide on the first number that is received
  - Since every correct process will receive the messages in the same order, they will all decide on the same value
- Solving Reliable Atomic Broadcast is equivalent to solving consensus

Broadcast System Model



- Important distinction between `send()` / `receive()` and `broadcast()` / `deliver()` primitives

Reliable Broadcast

Definition

A reliable broadcast is a broadcast which satisfies the following three properties:

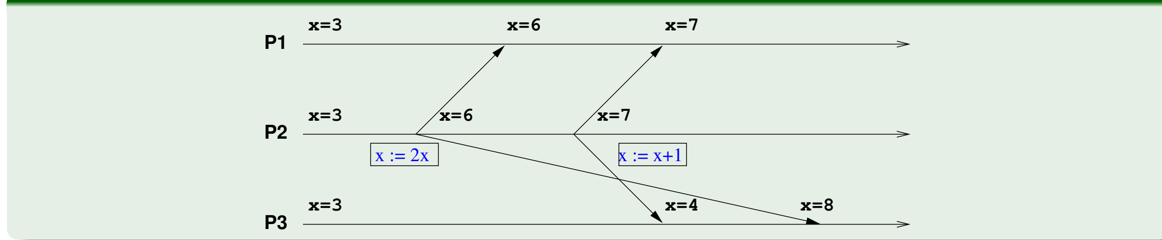
1. *Validity*: If a correct process broadcasts a message *m*, then all correct processes eventually deliver *m*.
2. *Agreement*: If a correct process delivers a message *m*, then all correct processes eventually deliver *m*.
3. *Integrity*: For any message *m*, every correct process delivers *m* at most once and only if *m* was previously broadcast by the sender of *m*.

Algorithm 1 Reliable broadcast algorithm

```
 1: p is the local process
 2: G is the group of processes

 3: procedure BROADCAST(R, m)              ▷ reliably broadcasts message m
 4:     for all g ∈ G do
 5:         send(g, m)
 6:     end for
 7: end procedure

 8: upon RECEIVE(m)
 9:     if p has not yet executed deliver(R, m) then
10:         for all g ∈ (G \ p) do
11:             send(g, m)
12:         end for
13:         deliver(R, m)                   ▷ deliver reliable broadcast message m
14:     fi
15: end upon
```

FIFO Broadcast

Example



Definition

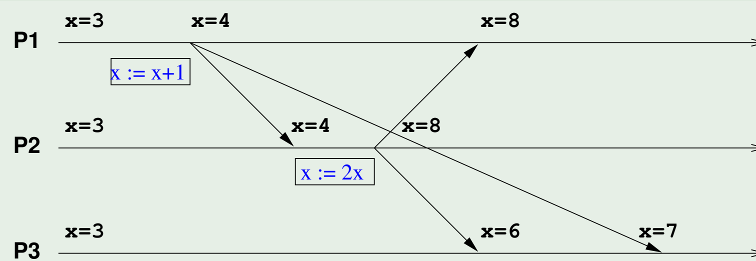
A broadcast is called a FIFO broadcast if the following condition holds: If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

Algorithm 2 FIFO reliable broadcast algorithm

```
 1: p is the local process
 2: G is the group of processes
 3: msgBag stores messages that are received out of order
 4: next[g] the next expected sequence number to be used by g
 5: seq the next sequence number used by process p

 6: procedure INITIALIZE
 7:     msgBag = ∅
 8:     seq ← 0
 9:     for all g ∈ G do
10:         next[g] ← 0
11:     end for
12: end procedure

13: procedure BROADCAST(F, m)              ▷ FIFO reliably broadcasts message m
14:     tag m with m.sender ← p
15:     tag m with m.seq ← seq
16:     seq ← seq + 1
17:     broadcast(R, m)
18: end procedure

19: upon DELIVER(R, m)
20:     q ← m.sender
21:     msgBag ← msgBag ∪ {m}
22:     while ∃n ∈ msgBag with n.sender = q and n.seq = next[q] do
23:         deliver(F, n)                  ▷ delivers FIFO reliable broadcast message m
24:         next[q] ← next[q] + 1
25:         msgBag ← msgBag \ n
26:     od
27: end upon
```

Causal Broadcast

Example

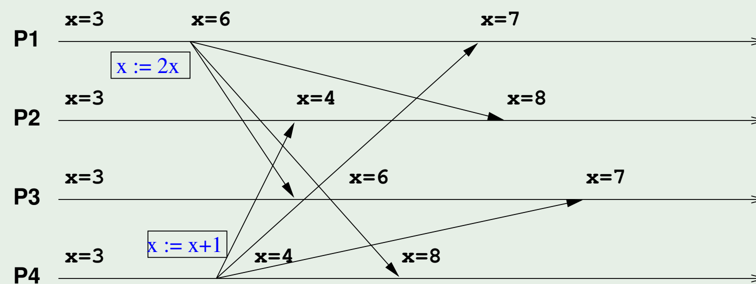


Definition

A broadcast is called a causal broadcast if the following condition holds: If a broadcast of a message m causally precedes the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m.

Atomic Broadcast

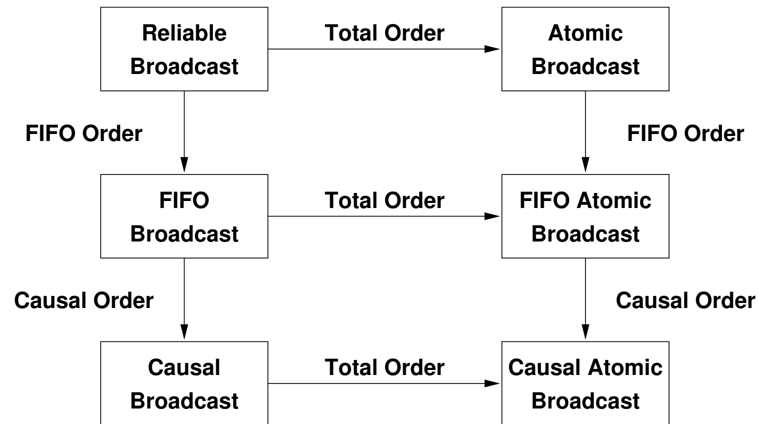
Example



Definition

A broadcast is called an atomic or totally ordered broadcast if the following condition holds: If correct processes p and q both deliver message m and m', then p delivers m before m' if and only if q delivers m before m'.

Broadcast Variants



Communicating Sequential Processes

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

CSP Notation: Events, Processes, Alphabet, Prefix

- Events are denoted by lower-case words (*coin*) or letters a, b, c ...
- Processes are denoted by upper-case words (*VMS*) or letters P, Q, R, ...
- Variables denoting events use lower-case letters x, y, z
- Variables denoting processes use upper-case letters X, Y, Z
- Sets of events are denoted by upper-case letters A, B, C
- The alphabet αP of a process P is the set of events it can react on.
- The process *STOP* is a process that does nothing, it never engages in any events.
- The process *RUN* is a process that engages in any event of its alphabet.
- Let x be an event and let P be a process. Then $(x \to P)$ describes a process which first engages in the event x and then behaves as described by P. We adopt the convention that the prefix $x \to P$ is right associative.

Events are considered to be atomic actions without duration. Actions that take time should be modelled by a start event and an end event. This is in particular important for actions where other events can happen while the action is being carried out.

Lets consider a simple vending machine VMS, which provides chocolate ($choc$) when customers insert a coin ($coin$). The alphabet of the process VMS is $\alpha VMS = \{coin, choc\}$.

A vending machine VMS consumes a coin before breaking:

$$(coin \to STOP)$$

A vending machine VMS serving two customers before breaking:

$$(coin \to (choc \to (coin \to (choc \to STOP))))$$

By leaving out the parenthesis, we get the following shorter notation:

$$(coin \to choc \to coin \to choc \to STOP)$$

Recursion and Choice

- A process description beginning with a prefix is said to be guarded.
- If $F(X)$ is a guarded expression containing the process name X and A is the alphabet of X, then the equation

$$X = F(X)$$

has a unique solution with the alphabet A. The solution of the expression is denoted as follows:

$$\mu X : A \bullet F(X)$$

- If x and y are distinct events, then

$$(x \rightarrow P \mid y \rightarrow Q)$$

describes a process which initially engages in either of the events x or y and then behaves as either P (if the first event was x) or Q (if the first event was y).

Consider a clock with the alphabet $\alpha CLOCK = \{tick\}$. A clock that emits a single tick afterwards behaves exactly like a clock:

$$CLOCK = (tick \rightarrow CLOCK)$$

By substitution, this recursive definition unfolds as shown below:

$$\begin{aligned} CLOCK &= (tick \rightarrow CLOCK) \\ &= (tick \rightarrow (tick \rightarrow CLOCK)) \\ &= (tick \rightarrow (tick \rightarrow (tick \rightarrow CLOCK))) \end{aligned}$$

The clock can be defined using the following expression:

$$CLOCK = \mu X : \{tick\} \bullet (tick \rightarrow X)$$

The simple vending machine (which does not break) is defined by:

$$VMS = \mu X : \{choin, choc\} \bullet (coin \rightarrow (choc \rightarrow X))$$

A vending machine $VMCC$ serving chocolate or coffee:

$$VMCC = \mu X : \{coin, choc, coffee\} \bullet (coin \rightarrow (choc \rightarrow X \mid coffee \rightarrow X))$$

Mutual recursion is possible if all right-hand sides of the equations are guarded and each unknown process appears exactly once on the left-hand side of one of the equations. Here is an example: A drink dispenser has a switch to make it dispense either orange or lemon. The switch setting is permanent, i.e., once set to orange, multiple orange drinks can be obtained.

$$\begin{aligned} \alpha DD = \alpha O = \alpha L &= \{setorange, setlemon, orange, lemon\} \\ DD &= (setorange \rightarrow O \mid setlemon \rightarrow L) \\ O &= (orange \rightarrow O \mid setlemon \rightarrow L \mid setorange \rightarrow O) \\ L &= (lemon \rightarrow L \mid setorange \rightarrow O \mid setlemon \rightarrow L) \end{aligned}$$

Basic Laws

$$STOP \neq (d \rightarrow P) \tag{L1A}$$

$$(c \rightarrow P) \neq (d \rightarrow Q) \quad \text{if } c \neq d \tag{L1B}$$

$$(c \rightarrow P \mid d \rightarrow Q) = (d \rightarrow Q \mid c \rightarrow P) \tag{L1C}$$

$$(c \rightarrow P) = (c \rightarrow Q) \equiv P = Q \tag{L1D}$$

$$(Y = F(Y)) \equiv (Y = \mu X \bullet F(X)) \quad \text{if } F(X) \text{ is a guarded expression} \tag{L2}$$

$$\mu X \bullet F(X) = F(\mu X \bullet F(X)) \tag{L2A}$$

The proof of (L1A), (L1B), (L1C), and (L1D) is trivial.

Traces

- A trace of a process is a finite sequence of symbols recording the events a process has engaged in up to some moment in time.
- A trace is denoted as a sequence of symbols, separated by commas and enclosed in angular brackets.
- The empty trace $\langle\rangle$ is the shortest trace of every possible process.
- Variables denoting traces are s, t, u
- Variables denoting sets of traces S, T, U
- Functions are denoted by f, g, h

Possible traces of the $VMCC$:

$\langle\rangle$ $\langle coin, choc, coin, choc\rangle$ $\langle coin, choc, coin, coffee\rangle$ $\langle coin, choc, coin\rangle$

Trace Catenation

- Given two traces s and t, the catenation of s and t is defined by $s^\frown t$.
- Basic catenation laws:

$$s^\frown\langle\rangle = \langle\rangle^\frown s = s \tag{L1}$$
$$s^\frown(t \frown u) = (s^\frown t)^\frown u \tag{L2}$$
$$s^\frown t = s^\frown u \equiv t = u \tag{L3}$$
$$s^\frown t = u^\frown t \equiv s = u \tag{L4}$$
$$s^\frown t = \langle\rangle \equiv s = \langle\rangle \wedge t = \langle\rangle \tag{L5}$$
$$t^0 = \langle\rangle \tag{L6}$$
$$t^{n+1} = t^\frown t^n \tag{L7}$$
$$t^{n+1} = t^{n\frown} t \tag{L8}$$
$$(s^\frown t)^{n+1} = s^\frown(t^\frown s)^{n\frown} t \tag{L9}$$

Example:

$$\langle coin, choc\rangle^\frown\langle coin, coffee\rangle = \langle coin, choc, coin, coffee\rangle$$

Trace Restriction

- The expression $(t \uparrow A)$ denotes the trace t restricted to symbols in the set A.
- Basic restriction laws:

$$\langle\rangle \uparrow A = \langle\rangle \tag{L1}$$

$$(s \frown t) \uparrow A = (s \uparrow A) \frown (t \uparrow A) \tag{L2}$$

$$\langle x \rangle \uparrow A = \langle x \rangle \quad \text{if } x \in A \tag{L3}$$

$$\langle x \rangle \uparrow A = \langle\rangle \quad \text{if } x \notin A \tag{L4}$$

Example:

$$\langle coin, choc, coin, choc\rangle \uparrow \{coin\} = \langle coin, coin\rangle \qquad \langle coin, choc, coin, choc\rangle \uparrow \{coffee\} = \langle\rangle$$

Trace Head and Tail

- The first symbol in a nonempty trace s is denoted s_0 (the head of trace s). The result of removing the first symbol from a nonempty trace s is denoted s' (the tail of trace s).
- Basic head and tail laws:

$$(\langle x \rangle)_0 = x \tag{L1}$$

$$(\langle x \frown s \rangle)' = s \tag{L2}$$

$$s = (\langle s_0 \rangle \frown s') \tag{L3}$$

$$s = t \equiv (s = t = \langle\rangle \vee (s_0 = t_0 \wedge s' = t')) \tag{L4}$$

Trace Star

- The set A^* is the set of all finite traces (including $\langle\rangle$) which are formed from symbols in the set A:
$$A^* = \{s | s \uparrow A = s\}$$
- Basic star laws:

$$\langle\rangle \in A^* \tag{L1}$$
$$\langle x \rangle \in A^* \equiv x \in A \tag{L2}$$
$$(s \frown t) \in A^* \equiv s \in A^* \wedge t \in A^* \tag{L3}$$
$$A^* = \{t | t = \langle\rangle \vee (t_0 \in A \wedge t' \in A^*)\} \tag{L4}$$

Trace Ordering

- An ordering relation $\leq$ on traces is defined as follows

$$s \leq t = (\exists u \bullet s^\frown u = t))$$

  and we say that s is a prefix of t.
- Basic ordering laws:

$$\langle\rangle \leq s \tag{L1}$$

$$s \leq s \tag{L2}$$

$$s \leq t \land t \leq s \Rightarrow s = t \tag{L3}$$

$$s \leq t \land t \leq u \Rightarrow s \leq u \tag{L4}$$

$$(\langle x\rangle^\frown s) \leq t \equiv t \neq \langle\rangle \land x = t_0 \land s \leq t' \tag{L5}$$

$$s \leq u \land t \leq u \Rightarrow s \leq t \lor t \leq s \tag{L6}$$

Trace Length

- The length of a trace t is denoted $\#t$. The number of occurrences of symbols from A in a trace t is defined as $t \uparrow A$. The number of occurances of a symbol x in a trace s is defined as $s \downarrow x = \#(s \uparrow \{x\})$.
- Basic length laws:

$$\#\langle\rangle = 0 \tag{L1}$$

$$\#\langle x\rangle = 1 \tag{L2}$$

$$\#(s^\frown t) = (\#s) + (\#t) \tag{L3}$$

$$\#(t \uparrow (A \cup B)) = \#(t \uparrow A) + \#(t \uparrow B) - \#(t \uparrow (A \cap B)) \tag{L4}$$

$$s \leq t \Rightarrow \#s \leq \#t \tag{L5}$$

$$\#(t^n) = n \times (\#t) \tag{L6}$$

Traces of a Process

- The functions $traces(P)$ returns the complete set of all possible traces of process P.
- If $s \in traces(P)$, then P/s (P after s) is a process which behaves as P from the time after P has engaged in all the actions recorded in the trace s.

$$\langle\rangle \in traces(P) \tag{L6}$$

$$s^\frown t \in traces(P) \implies s \in traces(P) \tag{L7}$$

$$traces(P) \subseteq (\alpha P)^* \tag{L8}$$

Laws of Traces of a Process

$$traces(STOP) = \{\langle\rangle\} \tag{L1}$$

$$traces(c \rightarrow P) = \{\langle\rangle\} \cup \{\langle c\rangle^\frown t \mid t \in traces(P)\} \tag{L2}$$

$$traces(c \rightarrow P \mid d \rightarrow Q) = \{\langle\rangle\} \cup \{\langle c\rangle^\frown t \mid t \in traces(P)\} \cup \{\langle d\rangle^\frown t \mid t \in traces(Q)\} \tag{L3}$$

$$traces(x : B \rightarrow P(X)) = \{\langle\rangle\} \bigcup_{b \in B} \{\langle b\rangle^\frown t \mid t \in traces(P(b))\} \tag{L4}$$

$$traces(\mu X : A \bullet F(X)) = \bigcup_{n \geq 0} traces(F^n(STOP)) \tag{L5}$$

Laws of Traces of a Process

- If s is a trace of process P, $s \in traces(P)$, then P/s (P after s) defines a process, which behaves the same as P behaves from the time after P has engaged in all the actions recorded in the trace s.
- Basic laws:

$$P/\langle\rangle = P \tag{L1}$$

$$P/(s^\frown t) = (P/s)/t \tag{L2}$$

$$(x : B \rightarrow P(x))/\langle c \rangle = P(c) \quad \text{if } c \in B \tag{L3}$$

$$(c \rightarrow P)/\langle c \rangle = P \tag{L3A}$$

$$traces(P/s) = \{t \mid s^\frown t \in traces(P)\} \quad \text{if } s \in traces(P) \tag{L4}$$

Example:

$$\begin{aligned} \alpha VMS &= \{coin, choc\} \\ VMS &= (coin \rightarrow choc \rightarrow VMS) \\ VMS/\langle coin \rangle &= (choc \rightarrow VMS) \end{aligned}$$

Specification, Satisfaction, Proof

- Let t denote an arbitrary trace of a process.
- A specification $S(t)$ is a predicate containing free variables over t.
- If a process P satisfies specification $S(t)$, we write P *sat* S.
- The goal is use our laws to proof P *sat* $S(t)$.

The number of chocolates dispensed must never exceed the number of coins inserted:

$$NOLOSS(t) = (\#(t \uparrow \{choc\}) \leq \#(t \uparrow \{coin\}))$$

The customer expects that the machine does not take multiple coins for a chocolate:

$$FAIR(t) = (\#(t \uparrow \{coin\}) \leq \#(t \uparrow \{choc\}) + 1)$$

The vendor has to make sure that both requirements are met:

$$VMSPEC(t) = NOLOSS(t) \wedge FAIR(t)$$

The designer of a vending machine has to ensure that the product meets the specification.

Satisfaction and Proof Laws

$$STOP\ sat\ (t = \langle\rangle) \quad \text{(L4A)}$$

$$P\ sat\ S(t) \Rightarrow (c \rightarrow P)\ sat\ (t = \langle\rangle \vee (t_0 = c \wedge S(t'))) \quad \text{(L4B)}$$

$$P\ sat\ S(t) \Rightarrow (c \rightarrow d \rightarrow P)\ sat\ (t \leq \langle c, d\rangle \vee (t \geq \langle c, d\rangle \wedge S(t''))) \quad \text{(L4C)}$$

$$P\ sat\ S(t) \wedge Q\ sat\ T(t) \Rightarrow (c \rightarrow P | d \rightarrow Q)\ sat\ (t = \langle\rangle \vee (t_0 = c \wedge S(t')) \vee (t_0 = d \wedge T(t'))) \quad \text{(L4D)}$$

$$\forall x : B \bullet (P(x)\ sat\ S(t, x)) \Rightarrow (x : b \rightarrow P(x))\ sat\ (t = \langle\rangle \vee (t_0 \in B \wedge S(t', t_0))) \quad \text{(L4)}$$

$$P\ sat\ S(t) \wedge s \in traces(P) \Rightarrow (P/s)\ sat\ S(s^\frown t) \quad \text{(L5)}$$

Lets prove that VMS satisfies $VMSPEC(t)$. We can write $VMSPEC(t)$ as follows:

$$S(t) = 0 \leq (\#(t \uparrow \{coin\}) - (\#(t \uparrow \{choc\}) \leq 1$$

We prove this by first looking at a machine that produces an empty trace, i.e., we have to show:

$$STOP\ sat\ S(\langle\rangle)$$

Since $\#(\langle\rangle \uparrow \{coin\}) = 0$ and $\#(\langle\rangle \uparrow \{choc\}) = 0$, we get $S(\langle\rangle) = 0 \leq 0 - 0 \leq 1)$, which is obviously true. Lets assume that $X\ sat\ S(t)$ and we now consider the process $(coin \rightarrow choc \rightarrow X)$. We need to show that $(coin \rightarrow choc \rightarrow X)\ sat\ S(t)$. Using the laws, we get:

$$\begin{aligned}(coin \rightarrow choc \rightarrow X)\ sat\ (t \leq \langle coin, choc\rangle \vee (t \geq \langle coin, choc\rangle \\ \wedge (0 \leq (\#(t'' \uparrow \{coin\} - \#(t'' \uparrow \{choc\})) \leq 1)) \\ \Rightarrow 0 \leq (\#(t \uparrow \{coin\}) - (\#(t \uparrow \{choc\}) \leq 1\end{aligned}$$

We trivially know that the following properties hold for all possible $t \leq \langle coin, choc\rangle$:

$$\begin{aligned}\#(\langle\rangle \uparrow \{coin\}) &= 0 \\ \#(\langle\rangle \uparrow \{choc\}) &= 0 \\ \#(\langle coin\rangle \uparrow \{choc\}) &= 0 \\ \#(\langle coin\rangle \uparrow \{coin\}) &= 1 \\ \#(\langle coin, choc\rangle \uparrow \{coin\}) &= 1 \\ \#(\langle coin, choc\rangle \uparrow \{choc\}) &= 1\end{aligned}$$

Interaction of Processes

- Two processes P and Q with the same alphabet ($\alpha P = \alpha Q$) interact in a lock-step way, denoted as $P \parallel Q$.
- Interaction means that both processes follow the same set of events.

Consider the process $VMCC$ that can serve chocolate or coffee. A greedy customer is taking a chocolate or a coffee if it is for free. If the greedy customer has to pay, then chocolate is preferred.

$$\begin{aligned} A &= \{coin, choc, coffee\} \\ VMCC &= \mu X : A \bullet (coin \rightarrow (choc \rightarrow X \mid coffee \rightarrow X)) \\ GRC &= \mu X : A \bullet (coffee \rightarrow GRC \mid choc \rightarrow GRC \mid coin \rightarrow choc \rightarrow GRC) \end{aligned}$$

When GRC and $VMCC$ interact, then GRC has no other choice than paying for chocolate since $VMCC$ will never dispense coffee or chocolate without first inserting a coin. Hence, we get the following interaction result:

$$(GRC \parallel VMCC) = \mu X : A \bullet (coin \rightarrow choc \rightarrow X)$$

Consider a generous customer GEC who leaves an extra coin after picking up a coffee:

$$GEC = \mu X : A \bullet (coin \rightarrow coffee \rightarrow coin \rightarrow X)$$

When the GEC and $VMCC$ interact, then GEC will pick a coffee and leave a coin but when GEC returns the system gets deadlocked since the $VMCC$ expects to dispense a drink while GEC expects to insert another coin.

$$(GEC \parallel VMCC) = \mu X : A \bullet (coin \rightarrow coffee \rightarrow coin \rightarrow STOP)$$

Interaction Laws

$$P \parallel Q = Q \parallel P \tag{L1}$$

$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \tag{L2}$$

$$P \parallel STOP = STOP \tag{L3A}$$
$$P \parallel RUN = P \tag{L3B}$$

$$(c \to P) \parallel (c \to Q) = (c \to (P \parallel Q)) \tag{L4A}$$
$$(c \to P) \parallel (d \to Q) = STOP \quad \text{if } c \neq d \tag{L4B}$$
$$(x : A \to P(x)) \parallel (y : B \to Q(y)) = (z : (A \cap B) \to (P(z) \parallel Q(z))) \tag{L4}$$

Example usage of the laws:

$$A = \{a, b, c\}$$
$$P = (a \to b \to P \mid b \to P)$$
$$Q = (a \to (b \to Q \mid c \to Q))$$

$$\begin{aligned}(P \parallel Q) &= a \to ((b \to P) \parallel (b \to Q \mid c \to Q)) \\ &= a \to (b \to (P \parallel Q)) \\ &= \mu X : A \bullet (a \to b \to X)\end{aligned}$$

Note that choices that do not “match” are dropped and that we finally can replace $P \parallel Q$ with X since the recursion is guarded.

Concurrency of Processes

- Two processes P and Q with different alphabets ($\alpha P \neq \alpha Q$) can execute concurrently, denoted as $P \parallel Q$.
- Events that are both in αP and αQ require simultaneous execution by P and Q.
- Events in αP that are not in αQ are of no concern for Q, and events in αQ that are not in αP are of no concern for P.
- The set of events that is possible for the concurrent combination of P and Q is given by

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

Consider a customer $CUST$ who prefers coffee. He utters a curse when there is only chocolate.

$$\alpha CUST = \{coin, choc, curse, coffee\}$$
$$CUST = (coin \rightarrow (coffee \rightarrow CUST \mid curse \rightarrow choc \rightarrow CUST))$$

Further, consider a vending machine NVM that produces a clink sound after inserting a coin and a clunk sound on completion of the transaction.

$$\alpha NVM = \{coin, clink, choc, clunk\}$$
$$NVM = (coin \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow NVM)$$

The combination of the two processes gives us a customer who always utters a curse. But we do not know whether this happens before or after the clink sound.

$$\alpha(NVM \parallel CUST) = \{coin, choc, curse, clink, clunk, coffee\}$$
$$(NVM \parallel CUST) = \mu X \bullet (coin \rightarrow (clink \rightarrow curse \rightarrow choc \rightarrow clunk \rightarrow X \mid curse \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow X))$$

Concurrency Laws

Let $a \in (\alpha P \setminus \alpha Q), b \in (\alpha Q \setminus \alpha P), \{c, d\} \subseteq (\alpha P \cap \alpha Q)$:

$$P \parallel Q = Q \parallel P \tag{L1}$$

$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \tag{L2}$$

$$P \parallel STOP = STOP \tag{L3A}$$

$$P \parallel RUN = P \tag{L3B}$$

$$(c \rightarrow P) \parallel (c \rightarrow Q) = (c \rightarrow (P \parallel Q)) \tag{L4A}$$

$$(c \rightarrow P) \parallel (d \rightarrow Q) = STOP \text{ if } c \neq d \tag{L4B}$$

$$(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q)) \tag{L5A}$$

$$(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q) \tag{L5B}$$

$$(a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q)) \mid b \rightarrow ((a \rightarrow P) \parallel Q)) \tag{L6}$$

Change of Symbols

- Sometimes it is convenient to change the symbols of a process or to derive another identical independent process by changing symbols.
- Let f be an injective function $f : \alpha P \rightarrow A$. We define the process $f(P)$ which engaged in the event $f(c)$ whenever P would engage in c:

$$\alpha f(P) = f(\alpha P)$$
$$traces(f(P)) = \{f^*(s) \mid s \in traces(P)\}$$

- $f^* : A \rightarrow B$ is derived from $f : A \rightarrow B$ and it maps a sequence of symbols in A^* to a sequence in B^* by applying f to each element of the sequence.

Labeled Processes

- Changing symbols allows us to create collections of similar processes which operate concurrently.
- We can use the technique to create labeled processes. A process P labeled by l is denoted by $l : P$. It engages in $l.x$ whenever P would engage in x.
- The function defining $l : P$ is $f_l(x) = l.x$ for all $x \in \alpha P$.

Suppose you have two vending machines standing side by side:

$$(left : VMCC) \parallel (right : VMCC)$$

Change of Symbols Laws

We will use $f(B) = \{f(x) \mid x \in B\}$, f^{-1} denotes the inverse of f, $f \circ g$ is the composition of f and g, f^* as defined above.

$$f(STOP) = STOP \tag{L1}$$
$$f(x : B \rightarrow P(x)) = (y : f(B) \rightarrow f(P(f^{-1}(y)))) \tag{L2}$$
$$f(P \parallel Q) = f(P) \parallel f(Q) \tag{L3}$$
$$f(\mu X : \bullet F(X)) = (\mu Y : f(A) \bullet f(F(f^{-1}(Y)))) \tag{L4}$$
$$f(g(P)) = (f \circ g)P \tag{L5}$$
$$traces(f(P)) = \{f^*(s) \mid s \in traces(P)\} \tag{L6}$$
$$f(P)/f^*(s) = f(P/s) \tag{L7}$$

The inverse function f^{-1} in L2 (and L4) converts $y \in f(B)$ (respectively $Y \in f(A)$) into a symbol of B (respectively A). Since we need f^{-1} here, f has to be injective.

Non-deterministic Choice

- If P and Q are processes with the same alphabet ($\alpha P = \alpha Q$), then the notation

$$P \sqcap Q$$

  denotes a process which behaves either like P or like Q.
- By construction, we have $\alpha(P \sqcap Q) = \alpha P = \alpha Q$.
- The decision whether the process $P \sqcap Q$ behaves like P or Q in made arbitrarily without knowledge or control by the environment.

Non-deterministic Choice Laws

$$P \sqcap P = P \tag{L1}$$
$$P \sqcap Q = Q \sqcap P \tag{L2}$$
$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \tag{L3}$$
$$x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q) \tag{L4}$$
$$(x : B \rightarrow (P(x) \sqcap Q(x))) = (x : B \rightarrow P(x)) \sqcap (x : B \rightarrow Q(x)) \tag{L5}$$
$$P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R) \tag{L6}$$
$$(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R) \tag{L7}$$
$$f(P \sqcap Q) = f(P) \sqcap f(Q) \tag{L8}$$

General Choice

- If P and Q are processes with the same alphabet ($\alpha P = \alpha Q$), then the notation

$$P \,\square\, Q$$

  denotes a process which behaves either like P or like Q.
- By construction, we have $\alpha(P \,\square\, Q) = \alpha P = \alpha Q$.
- The decision whether the process $P \,\square\, Q$ behaves like P or Q can be made by the environment. If the first action is only available for P, then P will be executed. If the first action is only available in Q, then Q will be executed. If the first action is possible in both P and Q, then the choice becomes non-deterministic.

In the special case where no initial event of P is possible in Q, the general choice operator $\square$ is the same as the choice operator |:

$$(c \rightarrow P) \,\square\, (d \rightarrow Q) = (c \rightarrow P \mid d \rightarrow Q) \quad \text{if } c \neq d$$

In the special case where the initial events of P and Q are the same, the general choice operator $\square$ is the same as the non-deterministic choice operator $\sqcap$:

$$(c \rightarrow P \,\square\, c \rightarrow Q) = (c \rightarrow P \sqcap c \rightarrow Q)$$

General Choice Laws

$$P \Box P = P \quad \text{(L1)}$$

$$P \Box Q = Q \Box P \quad \text{(L2)}$$

$$P \Box (Q \Box R) = (P \Box Q) \Box R \quad \text{(L3)}$$

$$P \Box STOP = P \quad \text{(L4)}$$

$$(x : A \rightarrow P(x)) \Box (y : B \rightarrow Q(y)) = \begin{cases} (z : (A \cup B) \rightarrow P(z)) & z \in (A \cup B) \\ (z : (A \cup B) \rightarrow Q(z)) & z \in (B \setminus A) \\ (z : (A \cup B) \rightarrow (P(z) \sqcap Q(z))) & z \in (A \cap B) \end{cases} \quad \text{(L5)}$$

$$P \Box (Q \sqcap R) = (P \Box Q) \sqcap (P \Box R) \quad \text{(L6)}$$

$$P \sqcap (Q \Box R) = (P \sqcap Q) \Box (P \sqcap R) \quad \text{(L7)}$$

Refusals

- Let X be a set of events which are offered initially by the environment of P. If P can deadlock on its first step when placed in this environment, then X is a refusal of P. The set of all such refusals of P is denoted *refusals*(P).
- If P is deterministic, then

$$(X \in \mathit{refusals}(P)) \equiv (X \cap P^0 = \{\})$$

  where $P^0 = \{x \mid \langle x \rangle \in \mathit{traces}(P)\}$.
- This can be generalized since the condition also applies to other steps of P. P is deterministic if

$$\forall s : \mathit{traces}(P) \bullet (X \in \mathit{refusals}(P/s) \equiv (X \cap (P/s)^0 = \{\})$$

The concept of a refusal permits a clear formal distinction to be made between deterministic and non-deterministic processes. A process is said to be deterministic if it can never refuse any event in which it can engage. In other words, a set is a refusal of a deterministic process only if that set contains no event in which that process can initially engage.

Refusals Laws

$$\begin{aligned}
\mathit{refusals}(\mathit{STOP}) &= \text{ all subsets of the alphabet} && \text{(L1)}\\
\mathit{refusals}(c \to P) &= \{X \mid X \subseteq (\alpha P \setminus \{c\})\} && \text{(L2)}\\
\mathit{refusals}(x : B \to P(x)) &= \{X \mid X \subseteq (\alpha P \setminus B)\} && \text{(L3)}\\
\mathit{refusals}(P \sqcap Q) &= \mathit{refusals}(P) \cup \mathit{refusals}(Q) && \text{(L4)}\\
\mathit{refusals}(P \,\Box\, Q) &= \mathit{refusals}(P) \cap \mathit{refusals}(Q) && \text{(L5)}\\
\mathit{refusals}(P \parallel Q) &- \{X \cup Y \mid X \in \mathit{refusals}(P) \land Y \in \mathit{refusals}(Q)\} && \text{(L6)}\\
\mathit{refusals}(f(P)) &= \{f(X) \mid X \in \mathit{refusals}(P)\} && \text{(L7)}\\
X \in \mathit{refusals}(P) &\implies X \subseteq \alpha P && \text{(L8)}\\
\{\} \in \mathit{refusals}(P) & && \text{(L9)}\\
(X \cup Y) \in \mathit{refusals}(P) &\implies X \in \mathit{refusals}(P) && \text{(L10)}\\
X \in \mathit{refusals}(P) &\implies (X \cup \{x\}) \in \mathit{refusals}(P) \lor \langle x \rangle \in \mathit{traces}(P) && \text{(L11)}
\end{aligned}$$

Concealment

- After constructing processes, we may want to conceal some internal events that were useful for the construction but which are irrelevant for the environment.
- If C is a finite set of events, then $P \setminus C$ is a process that behaves like P, except that each occurrence of any event in C is concealed.
- Obviously, we want $\alpha(P \setminus C) = (\alpha P) \setminus C$.

The noisy vending machine NVM may be placed in a soundproof box and we obtain a silent vending machine:

$$SVM = NVM \setminus \{clink, clunk\}$$

The mutual interactions of two concurrent processes can be seen as internal workings of the resulting process. Hence, it is usually the symbols in the intersection of the alphabets of the two processes that need to be concealed.

Interleaving

- If P and Q are processes with the same alphabet ($\alpha P = \alpha Q$), then the notation

$$P \,|||\, Q$$

  denotes concurrent execution of P and Q where common events are not processed simultaneously. Each action of the system is an action of exactly one of the processes.
- If one of the processes cannot engage in the action, then it must have been the other one.
- If both processes could have engaged in the same action, then the choice between them is non-deterministic.

A vending machine that will accept up to two coins before dispensing up to two chocolates:

$$VMS2 = (VMS \,|||\, VMS)$$

Note the difference between interaction ($P \parallel Q$) and interleaving ($P \,|||\, Q$) in the CSP framework. Interaction assumes that processes execute common events in a lock-step way while interleaving does not assume that common events are processed simultaneously. Compare how these two definitions differ:

$$VMS2 = (VMS \,|||\, VMS)$$

$$VMS2' = (VMS \parallel VMS)$$

Communication

- A communication event is described by a pair $c.v$ where c is the name of the channel on which the communication takes place and v is the value of the message which passes.
- The set of all messages which a process P can communicate on channel c is defined by:
$$\alpha c(P) = \{v \mid c.v \in \alpha P\}$$
- The functions $channel(c.v) = c$ and $message(c.v) = v$ provide us with the channel c and the message v of the communication event $c.v$.
- A process which writes v to c and then behaves list P is denoted as:
$$(c!v \to P) = (c.v \to P)$$
- A process which receives x on c and then behaves like $P(x)$ is denoted as:
$$(c?x \to P(x) = (y : \{y : channel(y) = c\} \to P(message(y)))$$

Channels are used for communication in only one direction and between only two processes.

If P and Q are two processes and c is an output channel of P and an input channel of Q, then the composition $(P \parallel Q)$ will have communication on c on each occasion where P outputs a message and Q simultaneously inputs that message.

Example: copy messages from left to right

$$\alpha left(COPY) = \alpha right(COPY)$$

$$COPY = (left?x \to right!x \to COPY)$$

Example: double numbers while copying from left to right

$$\alpha left(DOUBLE) = \alpha right(DOUBLE) = \mathbb{N}$$

$$DOUBLE = (left?x \to right!(x + x) \to DOUBLE)$$

Communication Choice

- Processes may need to communicate with a subset of a set of channels. To support this, the choice notation is adapted to channel names.
- If c and d are distinct channel names, then

$$(c?x \rightarrow P(x) \mid d?y \rightarrow Q(y))$$

denotes a process which initially inputs x on c and then behaves like $P(x)$ or initially inputs y on d and then behaves like $Q(y)$.
- The choice is determined by the channel that is ready first.

Example: merge messages from left1 and left2 to right

$$\alpha left1(MERGE) = \alpha left2(MERGE) = \alpha right(MERGE)$$

$$MERGE = (left1?x \rightarrow right!x \rightarrow MERGE \mid left2?x \rightarrow right!x \rightarrow MERGE)$$

Example: a fifo queue

$$\begin{aligned} BUFFER &= P_{\langle\rangle} \\ P_{\langle\rangle} &= left?x \rightarrow P_{\langle x\rangle} \\ P_{\langle x\rangle^\frown s} &= (left?y \rightarrow P_{\langle x\rangle^\frown s^\frown \langle y\rangle} \mid right!x \rightarrow P_s) \end{aligned}$$

Example: a stack

$$\begin{aligned} STACK &= P_{\langle\rangle} \\ P_{\langle\rangle} &= (empty \rightarrow P_{\langle\rangle} \mid left?x \rightarrow P_{\langle x\rangle}) \\ P_{\langle x\rangle^\frown s} &= (right!x \rightarrow P_s \mid left?y \rightarrow P_{\langle y\rangle^\frown \langle x\rangle^\frown s}) \end{aligned}$$

Communication Laws

$$(c!v \to P) \parallel (c?x \to Q(x)) = c!v \to (P \parallel Q(v)) \tag{L1}$$

$$((c!v \to P) \parallel (c?x \to Q(x))) \setminus C = (P \parallel Q(v)) \setminus C \text{ with } C = \{c.v \mid v \in \alpha c\} \tag{L2}$$

Note that a receiver is prepared to receive any value. Hence, communication is controlled by the sending process. Law L1 says that the receive event does not really matter, it is not observable from the outside. Law L2 says that the internal communication between two processes can be concealed.

Chaining (Pipes)

- Consider processes that have an input channel *left* and an output channel *right* and no other channels.
- Two such processes P and Q can be chained together so that the right channel of P is the left channel of Q and that the communication over the joint internal channel is concealed.
- The result of such a construction is denoted as $P \gg Q$.
- Chaining requires that certain constraints on the alphabet are met:

$$\alpha(P \gg Q) = \alpha left(P) \cup \alpha right(Q)$$
$$\alpha right(P) = \alpha left(Q)$$

Chaining is a popular mechanism in operating systems (e.g., Unix pipes).

Example: a pipe to multiply input values by four

$$\alpha left(DOUBLE) = \alpha right(DOUBLE) = \mathbb{N}$$
$$DOUBLE = (left?x \rightarrow right!(x + x) \rightarrow DOUBLE)$$

$$\alpha left(QUADRUPLE) = \alpha right(QUADRUPLE) = \mathbb{N}$$
$$QUADRUPLE = DOUBLE \gg DOUBLE$$

Chaining Laws

$$
\begin{aligned}
P \gg (Q \gg R) &= (P \gg Q) \gg R && \text{(L1)} \\
(right!v \to P) \gg (left?y \to Q(y)) &= P \gg Q(v) && \text{(L2)} \\
(right!v \to P) \gg (right!w \to Q) &= right!w \to ((right!v \to P) \gg Q) && \text{(L3)} \\
(left?x \to P(x) \gg (left?y \to Q(y)) &= left?x \to (P(x) \gg (left?y \to Q(y))) && \text{(L4)} \\
(left?x \to P(x)) \gg (right!w \to Q) &= (left?x \to (P(x) \gg (right!w \to Q) \\
&\quad \mid right!w \to ((left?x \to P(x)) \gg Q)) && \text{(L5)} \\
(left?x \to P(x)) \gg R \gg (right!w \to Q) &= (left?x \to (P(x) \gg R \gg (right!w \to Q) \\
&\quad \mid right!w \to ((left?x \to P(x)) \gg R \gg Q)) && \text{(L6)} \\
R \gg (right!w \to Q) &= right!w \to (R \gg Q) && \text{(L7)} \\
(left?x \to P(x)) \gg R &= left?x \to (P(x) \gg R) && \text{(L8)}
\end{aligned}
$$

Laws L3 and L4 enforce that external communication has precedence. L5 handles the case where both processes can engage in external communication. L6 extends this to the situation where additional processes exist in the chain.

Laws L7 and L8 are concerned with a chain where all processes start with output to the right or where all processes start with input from the left.

Part IV
Cryptography

This part introduces basic concepts of cryptography. The goal is to cover a minimum that is needed to understand how cryptography can be used later on to secure communication protocols or to more generally protect information. The material focuses on some currently widely used techniques but it is clear that cryptographic mechanisms change over time and hence some of the material may be more of a historic record in some 10-20 years from now.

Cryptography Primer

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Try to read the following text...

```
Jrypbzr gb Frpher naq Qrcraqnoyr Flfgrzf!

W!eslmceotmsey St oe lSbeacdunreep eaDn d

J!rfyzprbgzfrl Fg br yFornpqhaerrc rnQa q
```


The first line of ciphertext has been produced using the well-known ROT13 algorithm, a simple letter substitution cipher that replaces a letter with the 13th letter after it, in the alphabet. Given the 26 character basic Latin alphabet, ROT13 has the nice property that msg = ROT13(ROT13(msg)). ROT13 became popular in newsgroups of the 1980s in order to hide potentially offensive content. Applying ROT13 to the ciphertext

```
Jrypbzr gb Frpher naq Qrcraqnoyr Flfgrzf!
```

gives us the following cleartext message:

```
Welcome to Secure and Dependable Systems!
```

The second line of ciphertext has been produced by a simple permutation of the cleartext. By reading every uneven character and afterwards all remaining characters backwards, the ciphertext

```
W!eslmceotmsey St oe lSbeacdunreep eaDn d
```

turns into the following cleartext:

```
Welcome to Secure and Dependable Systems!
```

With this, it probably is easy to guess how the last line of ciphertext has been constructed: By applying both the ROT13 substitution and the permutation. This turns the ciphertext

```
J!rfyzprbgzfrl Fg br yFornpqhaerrc rnQa q
```

into the cleartext:

```
Welcome to Secure and Dependable Systems!
```

Terminology (Cryptography)

- *Cryptology* subsumes cryptography and cryptanalysis:
  - *Cryptography* is the art of secret writing.
  - *Cryptanalysis* is the art of breaking ciphers.
- *Encryption* is the process of converting *plaintext* into an unreadable form, termed *ciphertext*.
- *Decryption* is the reverse process, recovering the plaintext back from the ciphertext.
- A *cipher* is an algorithm for encryption and decryption.
- A *key* is some secret piece of information used as a parameter of a cipher and customizes the algorithm used to produce ciphertext.

It is important that the security of a cryptosystem rests on the secrecy of the keys and not on the secrecy if the algorithms. The algorithms of good cryptosystems should be publically known and withstand any attempts to break them.

Cryptosystem

Definition (cryptosystem)

A *cryptosystem* is a quintuple (M, C, K, E_k, D_k), where

- M is a cleartext space,
- C is a chiffretext space,
- K is a key space,
- $E_k : M \to C$ is an encryption transformation with $k \in K$, and
- $D_k : C \to M$ is a decryption transformation with $k \in K$.

For a given k and all $m \in M$, the following holds:

$$D_k(E_k(m)) = m$$

This definition is not yet complete. A cryptosystems must statisfy additional requirements since we do not want simple functions that are easy to revert.

Cryptosystem Requirements

- The transformations E_k and D_k must be efficient to compute.
- It must be easy to find a key $k \in K$ and the functions E_k and D_k.
- The security of the system rests on the secrecy of the key and not on the secrecy of the transformations (algorithms).
- For a given $c \in C$, it is difficult to systematically compute
  - D_k even if $m \in M$ with $E_k(m) = c$ is known
  - a cleartext $m \in M$ such that $E_k(m) = c$.
- For a given $c \in C$, it is difficult to systematically determine
  - E_k even if $m \in M$ with $E_k(m) = c$ is known
  - $c' \in C$ with $c' \neq c$ such that $D_k(c')$ is a valid cleartext in M.

We need to further formalize what "difficult to systematically determine" means. We need to express this in terms of complexity metrics.

Symmetric vs. Asymmetric Cryptosystems

Symmetric Cryptosystems

- Both (all) parties share the same key and the key needs to be kept secret.
- Examples: AES, DES (outdated), Twofish, Serpent, IDEA, ...

Asymmetric Cryptosystems

- Each party has a pair of keys: one key is public and used for encryption while the other key is private and used for decryption.
- Examples: RSA, DSA, ElGamal, ECC, ...

- For asymmetric cryptosystems, a key is a key pair (k, k^{-1}) where k denotes the public key and k^{-1} the associated private key.

The openssl command can be used to encrypt and decrypt data using a variety of different cryptosystems. To encrypt and decrypt a file using the symmetric cryptosystem AES in CBC mode with a key length of 256 bit, one can use the following shell commands:

```
echo 'Welcome to Secure and Dependable Systems!' > welcome.txt
openssl aes-256-cbc -in welcome.txt -out message.enc
openssl aes-256-cbc -d -in message.enc -out plaintext.txt
```

Cryptographic Hash Functions

Definition (cryptographic hash function)

A *cryptographic hash function* H is a hash function that meets the following requirements:

1. The hash function H is efficient to compute for arbitrary input m.
2. Given a hash value h, it should be difficult to find an input m such that $h = H(m)$ (preimage resistance).
3. Given an input m, it should be difficult to find another input $m' \neq m$ such that $H(m) = H(m')$ (2nd-preimage resistance).
4. It should be difficult to find two different inputs m and m' such that $H(m) = H(m')$ (collision resistance).

Cryptographic hash functions are used to compute a fixed size fingerprint, also called a message digest, of a variable length cleartext (message).

Cryptographic hashes can be computed with `openssl` command or using special purpose shell commands:

```
echo 'Welcome to Secure and Dependable Systems!' > welcome.txt
openssl dgst -sha256 welcome.txt
shasum -a 256 welcome.txt
```

Both commands calculate the hash value of the content of the file `welcome.txt` using the secure hash algorithm (SHA) and a hash value length of 256 bit. Note that the binary output is usually presented in hexadecimal notation (256 bit lead to 64 hexadecimal digits).

Digital Signatures

- Digital signatures are used to prove the authenticity of a message (or document) and its integrity.
  - The receiver can verify the claimed identity of the sender.
  - The sender can not deny that it did sent the message.
  - The receiver can not tamper with the message itself.
- Digitally signing a message (or document) means that
  - the sender puts a signature into a message (or document) that can be verified and
  - that we can be sure that the signature cannot be faked (e.g., copied from some other message)
- Digital signatures are often implemented by signing a cryptographic hash of the original message (or document) since this is usually less computationally expensive

Digital signatures are usually attached to the document that is signed. There are several standard formats for different use cases. The Cryptographic Message Syntax (CMS) defined in RFC 5652 can be used to sign arbitrary digital artefacts. A JSON signature format is defined in RFC 7515.

Usage of Cryptography

- Encrypting data in communication protocols (prevent eavesdropping)
- Encrypting data elements of files (e.g., passwords stored in a database)
- Encrypting entire files (prevent data leakage if machines are stolen or attacked)
- Encrypting entire file systems (prevent data leakage if machines are stolen or attacked)
- Encrypting backups stored on 3rd party storage systems
- Encrypting digital media to obtain revenue by selling keys (for example pay TV)
- Digital signatures of files to ensure that changes of file content can be detected or that the content of a file can be proven to originate from a certain source
- Encrypted token needed to obtain certain services or to authorize transactions
- Modern electronic currencies (cryptocurrency)

Cryptography is in wide-spread usage today and we are often not even aware of its usage. Cryptocurrencies like bitcoin and the underlying technology of blockchains have received much attention in the years 2016 and 2017.

Computer scientists and programmers need to be familiar with crypto APIs and they need to be sensitive to identify data in need of proper protection. Some rules of thumb:

- Never store credentials in cleartext.
- Always protect data during transmission.
- Always protect data when it is stored on third party computing and storage systems.
- If devices can be stolen, protect all data on such devices.
- There is very little you can trust, including yourself.

Symmetric Encryption Algorithms and Block Ciphers

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Substitution Ciphers

Definition (monoalphabetic and polyalphabetic substitution ciphers)

A *monoalphabetic substitution cipher* is a bijection on the set of symbols of an alphabet. A *polyalphabetic substitution cipher* is a substitution cipher with multiple bijections, i.e., a collection of monoalphabetic substitution ciphers.

- There are $|M|!$ different bijections of a finite alphabet M.
- Monoalphabetic substitution ciphers are easy to attack via frequency analysis since the bijection does not change the frequency of cleartext characters in the ciphertext.
- Polyalphabetic substitution ciphers are still relatively easy to attack if the length of the message is significantly longer than the key.

Lets represent all data as a number in $\mathbb{Z}_n$ (e.g., using ASCII code points or Unicode code points). Then we can consider monoalphabetic cryptosystems $(M = \mathbb{Z}_n, C = \mathbb{Z}_n, K = \mathbb{Z}, E_k, D_k)$ with

$$E_k(m) = (m + k) \bmod n$$
$$D_k(c) = (c - k) \bmod n$$

with $m \in M$, $c \in C$, and $k \in K$. This kind of cryptosystem is known as Caesar cipher. Historians believe that Gaius Julius Caesar used the monoalphabetic substitution cipher with the key $k = 3$ for the $n = 26$ latin characters.

The ROT13 cipher is essentially the monoalphabetic substitution cipher with the key $k = 13$ for the $n = 26$ latin characters (applied to lower-case and upper-case characters independently, leaving all other characters unchanged).

Lets represent all data as a number in $\mathbb{Z}_n$ (e.g., using ASCII code points or Unicode code points). Then we can consider monoalphabetic cryptosystems $(M = \mathbb{Z}_n, C = \mathbb{Z}_n, K = \mathbb{Z}^l, E_k, D_k)$ with

$$E_k(i, m) = (m + k_{(i \bmod l)}) \bmod n$$
$$D_k(i, c) = (c - k_{(i \bmod l)}) \bmod n$$

with $m \in M$, $c \in C$, and $k_i \in K$. The position i of the input symbol m in the cleartext (or the input symbol c in the ciphertext) determines which element of the key vector $k = (k_0, \ldots, k_{l-1})$ is used.

The Vinigére cipher splits a message into n blocks of a certain length l and then each symbol of a block is encrypted using a Caesar cipher with a different key k_i depending on the position of the symbol in the block. The Vinigére cipher, originally invented by Giovan Battista Bellaso in the 16th century, was once considered to be unbreakable, until Friedrich Kasiski published a general attack in the 19th century.

A notable special case of a polyalphabetic substitution cipher arises when the length of the key equals the length of the message and the key is only used once. In this case we call the cipher a one-time-pad.

Permutation Cipher

Definition (permutation cipher)

A *permutation cipher* maps a plaintext $m_0, \ldots, m_{l-1}$ to $m_{\tau(0)}, \ldots, m_{\tau(l-1)}$ where τ is a bijection of the positions $0, \ldots, l-1$ in the message.

- Permutation ciphers are also called transposition ciphers.
- To make the cipher parametric in a key, we can use a function τ_k that maps a key k to bijections.

An old permutation cipher is the rail-fence-cipher where a cleartext message is spelled out diagonally down and up over a number of rows and then read off row-by-row. The key k is the number of rows. Lets assume $k = 4$:

```
W     e     e     a     p     l     t
 e   m _   S c   _ n   e e   b e   s e
  l o   t _   u e   d D   n a   _ y   m !
   c     o     r     _     d     S     s

Weeapltem Sc neebeselot uedDna ym!cor dSs
```

A more general class of permutation ciphers are route ciphers where the plaintext is written out column-wise and then read back according to a specific pattern. Using $k = 5$ and reading row-by-row order:

```
Wm_rdels!
eeSe_net_
l_e_Dd_e_
ctcaeaSm_
oounpbys_

Wm_rdels!eeSe_net_l_e_Dd_e_ctcaeaSm_oounpbys_
```

Product Cipher

Definition (product cipher)

A *product cipher* combines two or more ciphers in a manner that the resulting cipher is more secure than the individual components to make it resistant to cryptanalysis.

- Combining multiple substitution ciphers results in another substitution cipher and hence is of little value.
- Combining multiple permutation ciphers results in another permutation cipher and hence is of little value.
- Combining substitution ciphers with permutation ciphers gives us ciphers that are much harder to break.

Chosen-Plaintext and Chosen-Ciphertext Attack

Definition (chosen plaintext attack)

In a *chosen-plaintext attack* the adversary can chose arbitrary cleartext messages m and feed them into the encryption function E to obtain the corresponding ciphertext.

Definition (chosen ciphertext attack)

In a *chosen-ciphertext attack* the adversary can chose arbitrary ciphertext messages c and feed them into the decryption function D to obtain the corresponding cleartext.

Polynomial and Negligible Functions

Definition (polynomial and negligible functions)

A function $f : \mathbb{N} \to \mathbb{R}^+$ is called

- *polynomial* if $f \in O(p)$ for some polynomial p
- *super-polynomial* if $f \notin O(p)$ for every polynomial p
- *negligible* if $f \in O(1/|p|)$ for every polynomial $p : \mathbb{N} \to \mathbb{R}^+$

Some closure properties:

- The sum of super-polynomial (polynomial, negligible) functions is super-polynomial (polynomial, negligible) again.
- The product of a super-polynomial (polynomial, negligible) function with a polynomial function is super-polynomial (polynomial, negligible) again.

Polynomial Time and Probabilistic Algorithms

Definition (polynomial time)

An algorithm A is called *polynomial time* if the worst-case time complexity of A for input of size n is a polynomial function.

Definition (probabilistic algorithm)

A *probabilistic algorithm* is an algorithm that may return different results when called multiple times for the same input.

Definition (probabilistic polynomial time)

A *probabilistic polynomial time* (PPT) algorithm is a probabilistic algorithm with polynomial time.

Fermat's little theorem states that if p is a prime number, then $x^p \equiv x \pmod{p}$ for any x. This relation is a necessary but not a sufficient condition for a prime number.

Require: $n > 3, k > 0$
1: **while** $k \neq 0$ **do**
2: $x \leftarrow random(2, n-2)$ ▷ pick x randomly in $[2, n-2]$
3: **if** $x^{(n-1)} \not\equiv 1 \pmod{n}$ **then**
4: **return 0** ▷ composite
5: **fi**
6: $k \leftarrow k - 1$
7: **end**
8: **return 1** ▷ probably prime

This algorithm is probabilistic:

- It needs randomness to work and may return different values for the same input.
- The results produced by the algorithm are probabilistic (probably prime) but if a number is found to be composite, then the result is correct.

The reasons why we choose $x \in \{2, \ldots, n-2\}$:

- Values outside $\mathbb{Z}_p$ are irrelevant because we take them modulo p anyway.
- The values $x = 0$ and $x = 1$ are useless because the property anyway holds for them.
- The value $x = p - 1$ is useless because the property holds anyway if p is odd.

One-way Functions

Definition (one-way function)

A function $f : \{0,1\}^* \to \{0,1\}^*$ is a *one-way function* if and only if f can be computed by a polynomial time algorithm, but any polynomial time randomized algorithm F that attempts to compute a pseudo-inverse for f succeeds with negligible probability.

- The existence of such one-way functions is still an open conjecture.
- Their existence would prove that the complexity classes P and NP are not equal.

One-way functions are super-polynomial hard to invert. Any algorithm A that attempts to guess an $x \in \{0,1\}^n$ such that y and $A(n,y)$ behave the same way under f suceeds with negligible probability.

Some functions that are commonly *believed* to be one-way functions are discrete exponentiation and multiplication. We do not know if there is a polynomial factoring algorithm.

Security of Ciphers

- What does it mean for an encryption scheme to be secure?
- Consider an adversary who can pick two plaintexts m_0 and m_1 and who randomly receives either $E(m_0)$ or $E(m_1)$.
- An encryption scheme can be considered secure if the adversary cannot distinguish between the two situations with a probability that is non-negligibly better than $1/2$.

Block Cipher

Definition (block cipher)

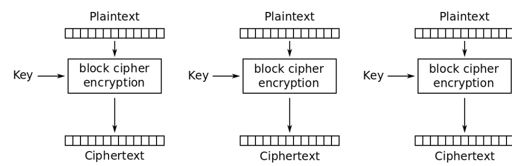
A *block cipher* is a cipher that operates on fixed-length groups of bits called a block.

- A given variable-length plaintext is split into blocks of fixed size and then each block is encrypted individually.
- The last block may need to be padded using zeros or random bits.
- Encrypting each block individually has certain shortcomings:
  - the same plaintext block yields the same ciphertext block
  - encrypted blocks can be rearranged and the receiver may not necessarily detect this
- Hence, block ciphers are usually used in more advanced modes in order to produce better results that reveal less information about the cleartext.

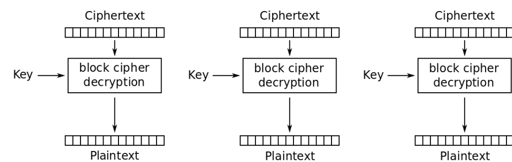
For further information:

- https://en.wikipedia.org/wiki/Block_cipher
- https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Electronic Code Book Mode



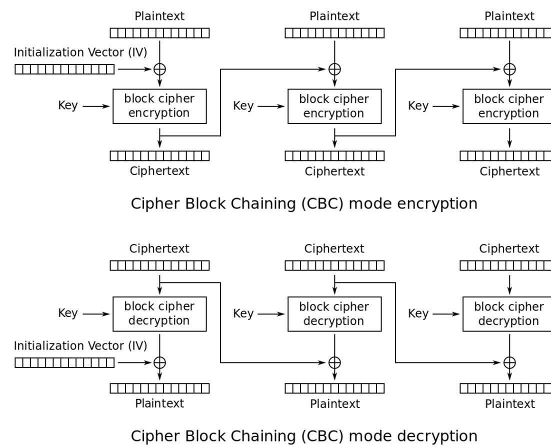
Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

- Encryption parallelizable: Yes
- Decryption parallelizable: Yes
- Random read access: Yes
- Lack of diffusion (does not hide data pattern)

Cipher Block Chaining Mode



Cipher Block Chaining (CBC) mode encryption

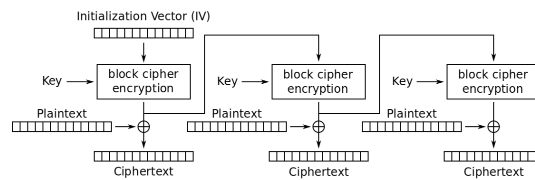
Cipher Block Chaining (CBC) mode decryption

- Encryption parallelizable: No
- Decryption parallelizable: Yes
- Random read access: Yes

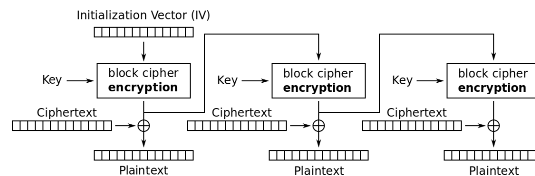
The initialization vector does not have to be secret but it needs to be random and it is ideally only used once. A random number only used once is called a nonce.

The sender has to communicate the initialization vector used to the receiver alongside the encrypted message. (An alternative is for the receiver to discard the first block of data.)

Output Feedback Mode



Output Feedback (OFB) mode encryption

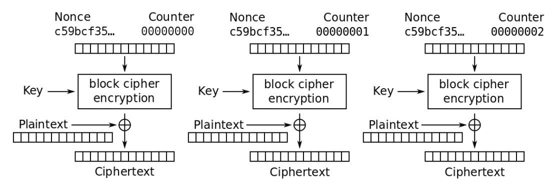


Output Feedback (OFB) mode decryption

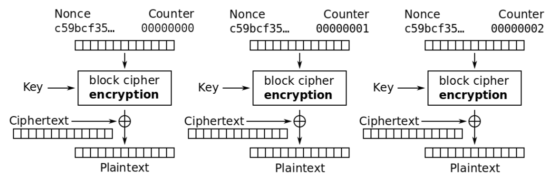
- Encryption parallelizable: No
- Decryption parallelizable: No
- Random read access: No

The output feedback mode of operation turns a block cipher into a stream cipher. A stream cipher is a symmetric key cipher where cleartext symbols are combined with a pseudorandom cipher stream (keystream). The chained block ciphers generate a keystream and the cleartext is XORed with the keys. Note that encryption and decryption work in exactly the same way in output feedback mode.

Counter Mode



Counter (CTR) mode encryption



Counter (CTR) mode decryption

- Encryption parallelizable: Yes
- Decryption parallelizable: Yes
- Random read access: Yes

The counter mode improves one of the shortcomings of output feedback mode, namely that it is sequential and does not support random access.

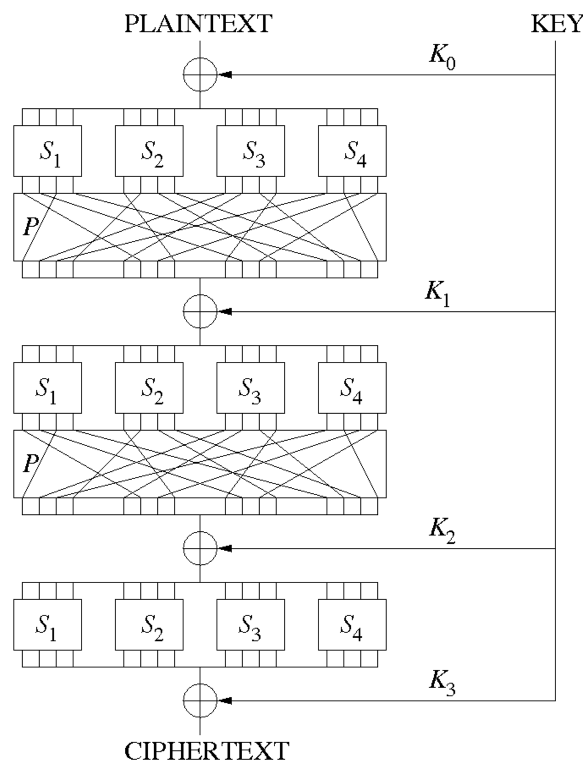
Substitution-Permutation Networks

Definition (substitution-permutation network)

A *substitution-permutation network* is a block cipher whose bijections arise as products of substitution and permutation ciphers.

- To process a block of N bits, the block is typically devided into b chunks of $n = N/b$ bits each.
- Each block is processed by a sequence of steps:
  - Substitution step: A chunk of n bits is substituted by applying a substitution box (S-box).
  - Permutation step: A permutation box (P-box) permutes the bits received from S-boxes to produce bits for the next round.
  - Key step: A key step maps a block by xor-ing it with a key.

A sketch of a substitution–permutation network with 3 rounds, encrypting a plaintext block of 16 bits into a ciphertext block of 16 bits.



For further information:

- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://en.wikipedia.org/wiki/Substitution%E2%80%93permutation_network

Advanced Encryption Standard (AES)

- Designed by two at that time relatively unknown cryptographers from Belgium (Vincent Rijmen and Joan Daemen, hence the name Rijndael of the proposal).
- Choosen by NIST (National Institute of Standards and Technology of the USA) after an open call for encryption algorithms.
- Characteristics:
  - overall blocksize: 128 bits
  - number of parallel S-boxes: 16
  - bitsize of an S-box: 8
  - 10 rounds with 128 bit keys
  - 12 rounds with 192 bit keys
  - 14 rounds with 256 bit keys

The Advanced Encryption Standard was published as a Federal Information Processing Standard (FIPS) by the National Institute of Standards and Technology (NIST) of the USA [11]. The algorithm can be implemented without any license fee requirements and it is very widely used these days. But note that in general the trust in encryption algorithms changes over time as new attacks are invented and technology evolves. Hence, it is crucial that cryptographic algorithms are replaceable, which is often called crypto agility.

Advanced Encryption Standard (AES) Rounds

- Round 0:
  (a) key step with k_0
- Round i: (i = 1, ..., r-1)
  (a) substitution step (called sub-bytes) with fixed 8-bit S-box (used 16 times)
  (b) permutation step (called shift-row) with a fixed permutation of 128 bits
  (c) substitution step (called mix-columns) with a fixed 32-bit S-box (used 4 times)
  (d) key step (called add-round-key) with a key k_i
- Round r: (no mix-columns)
  (a) substitution step (called sub-bytes) with fixed 8-bit S-box (used 16 times)
  (b) permutation step (called shift-row) with a fixed permutation of 128 bits
  (c) key step (called add-round-key) with a key k_r

The round keys $k_0, \ldots, k_r$ are generated by a key generator (also known as a key schedule) from the key k provided by the user of the algorithm.

For further information:

- https://en.wikipedia.org/wiki/Rijndael_key_schedule

Asymmetric Encryption Algorithms

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Asymmetric Encryption Algorithms

- Asymmetric encryption schemes work with a key pair:
  - a public key used for encryption
  - a private key used for decryption
- Everybody can send a protected message to a receiver by using the receiver's public key to encrypt the message. Only the receiver knowing the matching private key will be able to decrypt the message.
- Asymmetric encryption schemes give us a very easy way to digitally sign a message: A message encrypted by a sender with the sender's private key can be verified by any receiver using the sender's public key.
- Ron Rivest, Adi Shamir and Leonard Adleman (all then at MIT) published the RSA cryptosystem in 1978, which relies on the factorization problem of large numbers.
- Newer asynchronous cryptosystems often rely on the problem of finding discrete logarithms.

One inherent challenge associated with asymmetric encryption algorithms is the association of public keys with a certain identity. If Bob wants to send Alice an encrypted message, Bob first needs to obtain Alice's public key. If Mallory can interfere in this process and provide his public key instead of Alice's key, then Mallory will be able to read the message.

Another challenge associated with asymmetric encryption algorithms is the revocation of keys. If for some reason Alice has lost her private key, then the associated public key should not be used anymore and any data signed with Alice's private key should not be trusted anymore. Hence, there need to be mechanisms to revoke keys and to check whether a key has been revoked.

Rivest-Shamir-Adleman (RSA)

- Key generation:
  1. Generate two large prime numbers p and q of roughly the same length.
  2. Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
  3. Choose a number e satisfying $1 < e < \varphi(n)$ and $gcd(e, \varphi(n)) = 1$.
  4. Compute d satisfying $1 < d < \varphi(n)$ and $ed \bmod \varphi(n) = 1$.
  5. The public key is (n, e), the private key is (n, d); p, q and $\varphi(n)$ are discarded.
- Encryption:
  1. The cleartext m is represented as a sequence of numbers m_i with $m_i \in \{0, 1, \ldots, n-1\}$.
  2. Using the public key (n, e) compute $c_i = m_i^e \bmod n$ for all m_i.
- Decryption:
  1. Using the private key (n, d) compute $m_i = c_i^d \bmod n$ for all c_i.
  2. Transform the number sequence m_i back into the original cleartext m.

- Key generation:
  1. We choose the prime numbers $p = 47$ und $q = 71$.
  2. We compute $n = p \cdot q = 3337$ and $\varphi(n) = (p-1) \cdot (q-1) = 46 \cdot 70 = 3220$.
  3. We randomly choose $e = 79$ for which $gcd(79, 3220) = 1$.
  4. We compute $d = 1019$ satisfying $ed \bmod 3220 = 1$.
  5. The public key is $(3337, 79)$, the private key is $(3337, 1019)$.
- Encryption:
  1. The cleartext $m = 6882326879666683$ is devided into the cleartext blocks $m_i = [688, 232, 687, 966, 668, 3]$.
  2. Using the encryption key $(3337, 79)$, we compute

$$\begin{aligned}
c_1 &= 688^{79} \bmod 3337 = 1570 \\
c_2 &= 232^{79} \bmod 3337 = 2756 \\
c_3 &= 687^{79} \bmod 3337 = 2091 \\
c_4 &= 966^{79} \bmod 3337 = 2276 \\
c_5 &= 668^{79} \bmod 3337 = 2423 \\
c_6 &= 3^{79} \bmod 3337 = 158
\end{aligned}$$

     and we obtain the ciphertext blocks $c_i = [1570, 2756, 2091, 2276, 2423, 158]$.
- Decryption:
  1. Using the decryption key $(3337, 1019)$, we compute

$$\begin{aligned}
m_1 &= 1570^{1019} \bmod 3337 = 688 \\
m_2 &= 2756^{1019} \bmod 3337 = 232 \\
m_3 &= 2091^{1019} \bmod 3337 = 687 \\
m_4 &= 2276^{1019} \bmod 3337 = 966 \\
m_5 &= 2423^{1019} \bmod 3337 = 668 \\
m_6 &= 158^{1019} \bmod 3337 = 3
\end{aligned}$$

     and we obtain the cleartext blocks $m_i = [688, 232, 687, 966, 668, 3]$.

- Joining the blocks results in the cleartext $m = 6882326879666683$.

RSA Properties

- Security relies on the problem of factoring very large numbers.
- Quantum computers may solve this problem in polynomial time — so RSA will become obsolete once someone manages to build quantum computers.
- The prime numbers p and q should be at least 1024 (better 2048) bit long and not be too close to each other (otherwise an attacker can search in the proximity of $\sqrt{n}$).
- Since two identical cleartexts m_i and m_j would lead to two identical ciphertexts c_i and c_j, it is advisable to pad the cleartext numbers with some random digits.
- Large prime numbers can be found using probabilistic prime number tests.
- RSA encryption and decryption is compute intensive and hence usually used only on small cleartexts.

The RSA algorithm was protected by the U.S. Patent 4,405,829, which expired in September 2000.

There are various attempts to break RSA implementations. Typical problems encountered (and explored) are:

- Weak random number generators: If the random number generator used to create p and q is somewhat predictable, it becomes possible to reduce the search space.
- Timing attacks: If it is possible to measure the time it takes to decrypt known ciphertexts, then it is possible to find the decryption key d faster than applying brute force.

Cryptographic Hash Functions

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Cryptographic Hash Functions

- Cryptographic hash functions serve many purposes:
  - data integrity verification
  - integrity verification and authentication (via keyed hashes)
  - calculation of fingerprints for efficient digital signatures
  - adjustable proof of work mechanisms
- A cryptographic hash function can be obtained from a symmetric block encryption algorithm in cipher-block-chaining mode by using the last ciphertext block as the hash value.
- It is possible to construct more efficient cryptographic hash functions.

Cryptographic Hash Functions

Name	Published	Digest size	Block size	Rounds
MD-5	1992	128 b	512 b	4
SHA-1	1995	160 b	512 b	80
SHA-256	2002	256 b	512 b	64
SHA-512	2002	512 b	1024 b	80
SHA3-256	2015	256 b	1088 b	24
SHA3-512	2015	512 b	576 b	24

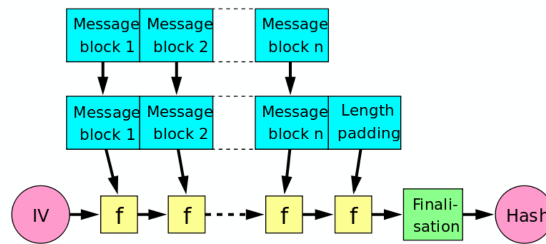
- MD-5 has been widely used but is largely considered insecure since the late 1990s.
- SHA-1 is largely considered insecure since the early 2000s.

Unix systems often came with command line tools to calculate hash values such as `sha1sum`, `sha256sum`, or `sha512sum`. The `openssl` command can also be used to calculate hash values.

```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ sha1sum welcome.txt
f200fd5c39f9a96647e4f4e80187eb8ee441a160  welcome.txt
$ openssl sha1 welcome.txt
SHA1(welcome.txt)= f200fd5c39f9a96647e4f4e80187eb8ee441a160
$ sha256sum welcome.txt
b34008c3d75d00108fb669366ebdb407b893ffbebdbd265e741fd62349db9868  welcome.txt
```

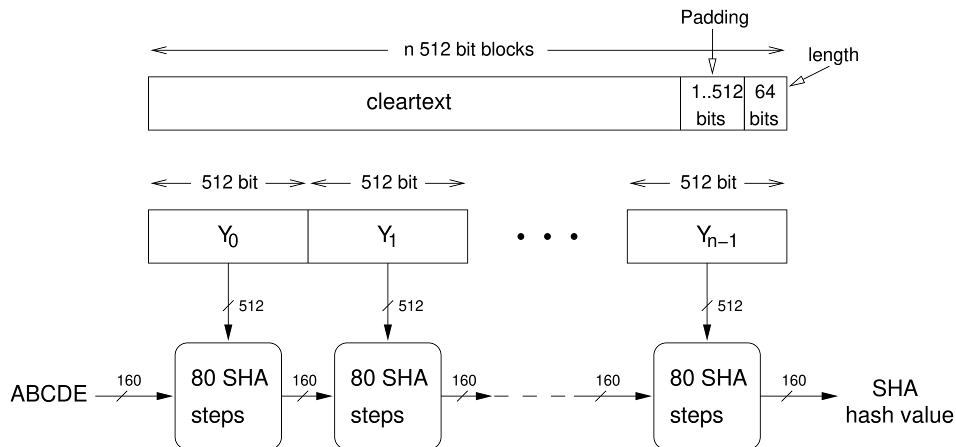
The `sha*sum` tools have been written to produce checksums for a list of files and to verify a list of files against previously computed checksums.

Merkle-Damgård Construction



- The message is padded and postfixed with a length value.
- The function f is a collision-resistant compression function, which compresses a digest-sized input from the previous step (or the initialization vector) and a block-sized input from the message into a digest-sized value.

Example (SHA-1):



Hashed Message Authentication Codes

- A keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key.
- An HMAC can be used to verify both data integrity and authenticity.
- An HMAC does not encrypt the message.
- The message must be sent alongside the HMAC hash. Parties with the secret key will hash the message again themselves, and if it is authentic, the received and computed hashes will match.

HMAC Computation

Given a key k, a hash function H, and a message m, the HMAC using H ($HMAC_H$) is calculated as follows:

$$HMAC_H(k, m) = H((k' \oplus opad) \parallel H((k' \oplus ipad) \parallel m))$$

- The key k' is derived from the original key k by padding k to the right with extra zeroes to the input block size of the hash function, or by hashing k if it is longer than that block size.
- The *opad* is the outer padding (0x5c5c5c...5c, one-block-long hexadecimal constant). The *ipad* is the inner padding (0x363636...36, one-block-long hexadecimal constant).
- The symbol $\oplus$ denotes bitwise exclusive or and the symbol $\parallel$ denotes concatenation.

HMACs [12] are widely used in communication protocols in situations where encryption of the messages is not considered important while message integrity and authentication of the messages is considered important. One reason is that using HMACs is computationally more efficient than using encryption algorithms.

The design of HMAC avoids attacks that are possible on simpler constructions:

- $HMAC_H = H(K||m)$ makes it easy for someone knowing m and the resulting $HMAC_H$ to append data to m leading to m' and to produce a valid $HMAC'_H$ for m' out of the HMAC of m.
- $HMAC_H = H(m||K)$ suffers from the problem that an attacker who can find a collision in the (unkeyed) hash function has a collision in the MAC.

However, given the increase of data collection and more efficient algorithms to do data correlation at large scale in recent years, there is a push to encrypt more and more data and with this the importance of HMACs in communication protocols may reduce in the future.

HMACs can be easily calculated on the command line using the `openssl` command:

```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ openssl sha1 -hmac "key" welcome.txt
HMAC-SHA1(welcome.txt)= 4c4cc66799b60777d96cc8dac3446d7103ab22ae
```

Authenticated Encryption with Associated Data

- It is often necessary to combine encryption with authentication of the data.
- Encryption protects the data and a message authentication code (MAC) protects the data against attempts to insert, remove, or modify data.
- Let E_k be an encryption function with key k and H_k a hash-based MAC with key k and $\|$ denotes concatenation.
- Encrypt-then-Mac (EtM)
$$E_k(M) \parallel H_k(E_k(M))$$
- Encrypt-and-Mac (EaM)
$$E_k(M) \parallel H_k(M)$$
- Mac-then-Encrypt (MtE)
$$E_k(M \parallel H_k(M))$$

TLS 1.3 supports only AEAD encryption algorithms. RFC 8446 [26] for example defines the cipher suites `TLS_AES_128_GCM_SHA256` or `TLS_AES_256_GCM_SHA384`. The AEAD algorithms are defined in RFC 5116 [19].

Digital Signatures and Certificates

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Digital Signatures

- Digital signatures are used to prove the authenticity of a message (or document) and its integrity.
  - Receiver can verify the claimed identity of the sender (authentiation)
  - The sender can later not deny that he/she sent the message (non-repudiation)
  - The message cannot be modified with invalidating the signature (integrity)
- A digital signature means that
  - the sender puts a signature into a message (or document) that can be verified and
  - that we can be sure that the signature cannot be faked (e.g., copied from some other message)
- Do not confuse digital signatures, which use cryptographic mechanisms, with electronic signatures, which may just use a scanned signature or a name entered into a form.

Digital Signatures using Asymmetric Cryptosystems

- Direct signature of a document m:
  - Signer: $S = E_{k^{-1}}(m)$
  - Verifier: $D_k(S) \stackrel{?}{=} m$
- Indirect signature of a hash of a document m:
  - Signer: $S = E_{k^{-1}}(H(m))$
  - Verifier: $D_k(S) \stackrel{?}{=} H(m)$
- The verifier needs to be able to obtain the public key k of the signer from a trustworthy source.
- The signature of a hash is faster (and hence more common) but it requires to send the signature S along with the document m.

Public Key Certificates

Definition (public key certificate)

A *public key certificate* is an electronic document used to prove the ownership of a public key. The certificate includes

- information about the public key,
- information about the identity of its owner (called the subject), and
- the digital signature of an entity that has verified the certificate's contents (called the issuer).

- If the signature is valid, and the software examining the certificate trusts the issuer of the certificate, then it can trust the public key contained in the certificate to belong to the subject of the certificate.

Public Key Infrastructure (PKI)

Definition

A *public key infrastructure* (PKI) is a set of roles, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption.

- A central element of a PKI is the certificate authority (CA), which is responsible for storing, issuing and signing digital certificates.
- CAs are often hierarchically organized. A root CA may delegate some of the work to trusted secondary CAs if they execute their tasks according to certain rules defined by the root CA.
- A key function of a CA is to verify the identity of the subject (the owner) of a public key certificate.

X.509 Certificate ASN.1 Definition

```
Certificate  ::=  SEQUENCE  {
     tbsCertificate       TBSCertificate,
     signatureAlgorithm   AlgorithmIdentifier,
     signatureValue       BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     extensions      [3]  EXPLICIT Extensions OPTIONAL
                          -- If present, version MUST be v3
     }
```


The widely used standard for public key certificates goes back to work done by ITU-T in the late 1980s to define open standards for directory services. The directory standard was known under the name X.500 and X.509 was its public key certificate format. Back in the late 1980, it was popular to define the format of messages using the Abstract Syntax Notation One (ASN.1). The ASN.1 type definition

```
Certificate  ::=  SEQUENCE  {
   tbsCertificate       TBSCertificate,
   signatureAlgorithm   AlgorithmIdentifier,
   signatureValue       BIT STRING  }
```

has to be read as if it would define the following C structure:

```
typedef struct {
    TBSCertificate      tbsCertificate;
    AlgorithmIdentifier signatureAlgorithm;
    BitString           signatureValue;
} Certificate;
```

In other words, a Certificate is composed of a structure that holds information about the subject and the certificate (the TBSCertificate) and a signature and a signature algorithm identifier. The important fields of the TBSCertificate are:

- version: The version of the encoded certificate. The current version is version 3.
- serialNumber: A unique positive integer assigned by the CA to each certificate.
- signature: The algorithm identifier for the algorithm used by the CA to sign the certificate.
- issuer: The issuer field identifies the entity that has signed and issued the certificate.
- validity: The certificate validity period is the time interval during which the CA warrants that it will maintain information about the status of the certificate.
- subject: The subject field identifies the entity associated with the public key stored in the subject public key field.
- subjectPublicKeyInfo: This field is used to carry the public key together with an identification of the algorithm with which the key is to be used (e.g., RSA).

X.509 Certificate ASN.1 Definition

```
Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER

Validity ::= SEQUENCE {
     notBefore      Time,
     notAfter       Time }

Time ::= CHOICE {
     utcTime        UTCTime,
     generalTime    GeneralizedTime }

UniqueIdentifier  ::=  BIT STRING

SubjectPublicKeyInfo  ::=  SEQUENCE  {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING  }

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
                 -- contains the DER encoding of an ASN.1 value
                 -- corresponding to the extension type identified
                 -- by extnID
     }
```


Implementations often store certificates in .crt files. Such a file can be converted into human readable text using an `openssl` shell command:

```
$ openssl x509 -in grader.eecs.jacobs-university.de.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            1c:56:75:9d:a9:94:a4:11:70:ea:b1:e7
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = DE, O = Jacobs University Bremen gGmbH, OU = IRC-IT, CN = Jacobs University CA - G01
        Validity
            Not Before: Nov 24 14:24:14 2016 GMT
            Not After : Jul  9 23:59:00 2019 GMT
        Subject: C = DE, ST = Bremen, L = Bremen, O = Jacobs University Bremen gGmbH, CN = grader.eecs.jacobs-university.de
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ae:7a:f1:e4:8c:2e:70:a6:8a:b9:ec:74:df:7e:
                    06:87:e6:eb:6c:c0:3b:62:35:40:8b:dd:cc:c3:28:
                    c2:2d:32:30:4f:08:e1:bb:40:c1:52:84:cb:5a:90:
                    23:74:9d:cc:13:88:9a:55:9c:55:57:18:90:88:af:
                    fa:8e:2a:17:15:68:a2:e8:e9:6a:9e:2e:4b:aa:5a:
                    d0:2b:c9:f3:ba:0d:08:3e:28:6c:64:fd:fb:3e:b9:
                    82:54:c7:f6:6c:9a:67:64:7a:de:c5:5e:bb:51:ce:
                    5c:24:f0:57:d5:11:12:e8:2e:c5:02:c0:a2:da:f3:
                    17:69:ba:de:c0:e1:df:cf:86:62:e0:fb:4e:18:19:
                    eb:cf:53:32:67:28:64:ae:b8:d2:bf:d6:9c:55:0a:
                    92:b6:49:df:7e:91:47:13:b8:a5:58:7a:ec:45:38:
                    88:d4:77:42:5c:d0:d4:77:b5:c5:79:01:b0:94:ea:
                    de:3b:5f:da:75:07:50:9e:25:14:53:c7:18:91:16:
                    75:2a:1c:30:21:64:a8:43:e5:f6:6b:56:27:6e:bc:
                    4f:c8:56:97:c5:1f:13:b8:9c:dd:e2:74:7e:cb:8d:
                    67:29:81:96:59:5b:18:6a:02:c5:20:a7:5f:23:5d:
                    de:3a:7d:64:38:28:45:52:ce:e9:2f:03:f9:52:ae:
                    1c:a3
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Certificate Policies:
                Policy: 1.3.6.1.4.1.22177.300.1.1.4.3.5
                Policy: 1.3.6.1.4.1.22177.300.2.1.4.3.1
                Policy: 1.3.6.1.4.1.22177.300.1.1.4
                Policy: 1.3.6.1.4.1.22177.300.30
```

```
            Policy: 2.23.140.1.2.2

        X509v3 Basic Constraints:
            CA:FALSE
        X509v3 Key Usage: critical
            Digital Signature, Key Encipherment
        X509v3 Extended Key Usage:
            TLS Web Server Authentication
        X509v3 Subject Key Identifier:
            7D:A1:33:72:42:86:A8:93:15:06:B2:FF:03:FD:12:EC:5F:A4:4E:61
        X509v3 Authority Key Identifier:
            keyid:1C:AB:41:DA:D4:95:D4:9D:1E:82:CD:4D:7D:13:25:37:EC:1F:88:A7

        X509v3 Subject Alternative Name:
            DNS:grader.eecs.jacobs-university.de
        X509v3 CRL Distribution Points:

            Full Name:
              URI:http://cdp1.pca.dfn.de/jacobs-university-ca/pub/crl/cacrl.crl

            Full Name:
              URI:http://cdp2.pca.dfn.de/jacobs-university-ca/pub/crl/cacrl.crl

        Authority Information Access:
            OCSP - URI:http://ocsp.pca.dfn.de/OCSP-Server/OCSP
            CA Issuers - URI:http://cdp1.pca.dfn.de/jacobs-university-ca/pub/cacert/cacert.crt
            CA Issuers - URI:http://cdp2.pca.dfn.de/jacobs-university-ca/pub/cacert/cacert.crt

Signature Algorithm: sha256WithRSAEncryption
     4a:8a:c1:33:d1:5a:0d:6e:a1:c4:90:85:0a:3e:db:44:f2:b2:
     95:31:12:31:6f:5a:a5:a8:34:1c:91:39:0a:cd:03:e6:7e:9d:
     4e:ee:a7:16:29:9b:24:1c:b9:e3:c9:fd:7a:1e:f3:02:92:cb:
     46:41:05:ca:82:4f:5a:39:a6:41:9d:76:27:61:2f:1f:de:44:
     91:af:48:4e:91:07:22:bd:09:1a:94:74:59:8c:29:43:72:b7:
     0f:37:3c:b2:b4:1d:7b:8d:96:d7:d5:a0:1b:6a:b3:8b:b8:f2:
     88:ee:39:a0:8e:76:bf:3d:f1:a4:0a:22:43:f4:ab:e9:df:4b:
     a0:1e:b5:37:71:28:e9:38:ca:e0:61:63:fb:32:51:34:7b:d7:
     3d:6b:7b:be:9a:57:1b:6c:c2:5b:f2:80:12:25:39:26:20:e0:
     af:c2:b0:d9:ec:36:cf:33:b0:0f:22:de:70:e4:11:c1:56:d3:
     73:9b:12:e3:02:61:19:64:99:6d:dd:8a:fe:58:72:a0:f4:18:
     91:06:69:58:c1:05:d0:16:c3:e4:8e:70:72:56:7d:28:9f:60:
     a6:32:21:69:b7:64:07:fe:19:49:f1:f1:58:dc:4c:dd:a2:f4:
     b5:7a:f3:f8:f4:62:f5:89:ee:aa:9c:fd:64:d5:6d:12:48:cf:
     3e:5e:70:dc
```

X.509 Subject Alternative Name Extension

```
id-ce-subjectAltName OBJECT IDENTIFIER ::=  { id-ce 17 }

SubjectAltName ::= GeneralNames

GeneralNames ::= SEQUENCE SIZE (1..MAX) OF GeneralName

GeneralName ::= CHOICE {
     otherName                       [0]     OtherName,
     rfc822Name                      [1]     IA5String,
     dNSName                         [2]     IA5String,
     x400Address                     [3]     ORAddress,
     directoryName                   [4]     Name,
     ediPartyName                    [5]     EDIPartyName,
     uniformResourceIdentifier       [6]     IA5String,
     iPAddress                       [7]     OCTET STRING,
     registeredID                    [8]     OBJECT IDENTIFIER }

OtherName ::= SEQUENCE {
     type-id    OBJECT IDENTIFIER,
     value      [0] EXPLICIT ANY DEFINED BY type-id }

EDIPartyName ::= SEQUENCE {
     nameAssigner            [0]     DirectoryString OPTIONAL,
     partyName               [1]     DirectoryString }
```


The subject of a X.509 certificate is a so called Distinguished Name. While this format made sense in the X.500 world, we usually use other names in the Internet context. The Subject Alternative Name Extension provides a mechanism to have an extensible format for alternative names. In the example shown on the previous pages, the subject alternative name is a DNS name and the value is `grader.eecs.jacobs-university.de`.

Some organizations make use of wildcard certificates where one DNS label (a part of a DNS name) may include a wildcard character (*). A single wildcard certificate for `*.example.com` will secure all subdomains, such as `payment.example.com`, `contact.example.com`, or `www.example.com`. While wildcard certificates may be convenient for system administrators, it is generally recommended to not use them, see for example Section 7.2 of RFC 6125.

Key Exchange Schemes

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Cryptographic Protocol Notation

$A, B, \ldots$	principals
$K_{AB}, \ldots$	symmetric key shared between A and B
$K_A, \ldots$	public key of A
$K_A^{-1}, \ldots$	private key of A
H	cryptographic hash function
$N_A, N_B, \ldots$	nonces (fresh random messages) chosen by A, B, ...
P, Q, R	variables ranging over principals
X, Y	variables ranging over statements
K	variable over a key
$\{m\}_K$	message m encrypted with key K

Key Exchange and Ephemeral Keys

Definition (key exchange)

Key exchange (also key establishment) is any method by which cryptographic keys are exchanged between two parties, allowing use of a cryptographic algorithm.

- Key exchange methods are important to establish ephemeral keys even if two principals have already access to suitable long-term keys
- Ephemeral keys help to protect keys that are used to bootstrap secure communication between principals
- Ephemeral keys can provide perfect forward secrecy

Ephemeral keys are desirable since keys become easier to break the more they are used. Hence, long-term asymmetric keys (that are used with a public key infrastructure) are typically not used directly to encrypt data. They are instead used to establish ephemeral keys between communicating parties.

Perfect forward secrecy assures that ephemeral keys will not be compromised even if long-term keys of the communicating parties are compromised in the future.

Diffie-Hellman Key Exchange

- Initialization:
  - Define a prime number p and a primitive root g of $\mathbb{Z}_p$ with $g < p$. The numbers p and g can be made public.
- Exchange:
  - A randomly picks $x_A \in \mathbb{Z}_p$ and computes $y_A = g^{x_A} \bmod p$. x_A is kept secret while y_A is sent to B.
  - B randomly picks $x_B \in \mathbb{Z}_p$ and computes $y_B = g^{x_B} \bmod p$. x_B is kept secret while y_B is sent to A.
  - A computes:

$$K_{AB} = y_B^{x_A} \bmod p = (g^{x_B} \bmod p)^{x_A} \bmod p = g^{x_A x_B} \bmod p$$

  - B computes:

$$K_{AB} = y_A^{x_B} \bmod p = (g^{x_A} \bmod p)^{x_B} \bmod p = g^{x_A x_B} \bmod p$$

  - A and B now own a shared key K_{AB}.

The Diffie-Hellman key exchange [9] uses discrete exponentiation $b^x \bmod m$, which is fast to compute even for large exponents x. For the inverse function, the discrete logarithm, there is no known efficient algorithm (a probabilistic algorithm with polynomial time). The Diffie-Hellman key exchange uses the multiplicative group $\mathbb{Z}_p$ of integers modulo p, where p is prime.

The value g is a primitve root of $\mathbb{Z}_p$ if the expression $g^t \bmod p$ for $t \in \{0, 1, 2, \ldots, p-2\}$ results in the numbers $\{1, 2, \ldots, p-1\}$ (in any order).

```haskell
import Data.List

-- check whether p is a prime number
isPrime :: Integer -> Bool
isPrime p = null [ x | x <- [2..p-1], p `mod` x == 0]

-- check whether g is a primitive root of Z_p.
isRoot :: Integer -> Integer -> Bool
isRoot p g = (sort xs) == [1..p-1]
    where xs = map (\x -> g^x `mod` p) [0..p-2]
```

Example:

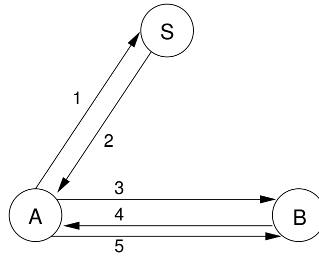
- A and B agree to use the prime number $p = 47$ and the primitive root $g = 5$
- A picks $x_A = 18$ and computes $y_A = 5^{18} \bmod 47 = 2$
- B picks $x_B = 22$ and computes $y_B = 5^{22} \bmod 47 = 28$
- A sends $y_A = 2$ to B
- B sends $y_B = 28$ to A
- A computes $K_{AB} = y_B^{x_A} \bmod p = 28^{18} \bmod 47 = 24$
- B computes $K_{AB} = y_A^{x_B} \bmod p = 2^{22} \bmod 47 = 24$

The Diffie-Hellman exchange can be attacked easily if an attacker can act as a man-in-the-middle. Hence, the usage of the Diffie-Hellman exchange requires that the communicating parties can verify that there is no man-in-the-middle involved (e.g., by protecting the exchange using long-term keys).

Diffie-Hellman Key Exchange (cont.)

- A number g is a primitive root of $\mathbb{Z}_p = \{0, \ldots, p-1\}$ if the sequence $g^1 \bmod p, g^2 \bmod p, \ldots, g^{p-1} \bmod p$ produces the numbers $1, \ldots, p-1$ in any permutation.
- p should be choosen such that $(p-1)/2$ is prime as well.
- p should have a length of at least 2048 bits.
- Diffie-Hellman is not perfect: An attacker can play "man in the middle" (MIM) by claiming B's identity to A and A's identity to B.

Needham-Schroeder Protocol

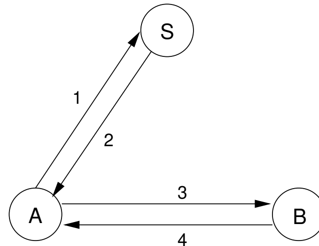


Msg 1: $A \rightarrow S :$ A, B, N_a
Msg 2: $S \rightarrow A :$ $\{N_a, B, K_{AB}, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
Msg 3: $A \rightarrow B :$ $\{K_{AB}, A\}_{K_{BS}}$
Msg 4: $B \rightarrow A :$ $\{N_b\}_{K_{AB}}$
Msg 5: $A \rightarrow B :$ $\{N_b - 1\}_{K_{AB}}$

The Needham-Schroeder protocol [22] assumes that the two principals A and B both share a key with the server S

- Principals A and B both share a secrect (K_{AS}, K_{BS}) key with an authentication server S.
- A and B need a shared key to secure communication between them.
- Idea: The authentication server creates a key K_{AB} and distributes it to the principals A and B, protected by the keys shared with S.
- Principal B must believe in the freshness of K_{AB} in the third message. This allows an attacker to break K_{AB} without any time constraint.
- The problem can be solved by introducing time stamps. However, timestamps require securely synchronized clocks.
- The double encryption in the second message is redundant.
- How can we find protocol shortcomings in a structured way?

Kerberos Protocol



Msg 1: $A \rightarrow S:$ A, B

Msg 2: $S \rightarrow A:$ $\{T_s, L, K_{AB}, B, \{T_s, L, K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$

Msg 3: $A \rightarrow B:$ $\{T_s, L, K_{AB}, A\}_{K_{BS}}, \{A, T_a\}_{K_{AB}}$

Msg 4: $B \rightarrow A:$ $\{T_a + 1\}_{K_{AB}}$

The Kerberos authentication service was developed at MIT. Version 5 of Kerberos is defined in RFC 4120 [23]. (RFC 6649 [2] deprecates weak crytographic algorithms in Kerberos.)

- Improved version of the Needham-Schroeder protocol.
- Uses time stamps to address the flaw in the original Needham-Schroeder protocol.
- Uses only four messages instead five.
- Can Needham-Schroeder be fixed without introducing time stamps?
- If yes, can it be done in just four messages, or are five or even more messages required?

For an alternate solution that does not require synchronized cloks, see [16].

Kerberos has been implemented as the authentication protocol in Microsoft's Active Directory.

BAN Logic

- Idea: Use a formal logic to reason about authentication protocols.
- Answer questions such as:
  - What can be achieve with the protocol?
  - Does a given protocol have stronger prerequisites than some other protocol?
  - Does a protocol do something which is not needed?
  - Is a protocol minimal regarding the number of messages exchanged?
- The Burrows-Abadi-Needham (BAN) logic was a first attempt to provide a formalism for authentication protocol analysis.
- The spi calculus, an extension of the pi calculus, was introduced later to analyze cryptographic protocols.

The BAN logic appeared in 1989 [4] and the spi calculus about ten years later in 1999 [1].

Using BAN Logic

- Steps to use BAN logic:
  1. Idealize the protocol in the language of the formal BAN logic.
  2. Define your initial security assumptions in the language of BAN logic.
  3. Use the productions and rules of the logic to deduce new predicates.
  4. Interpret the statements you've proved by this process. Have you reached your goals?
  5. Trim unnecessary fat from the protocol, and repeat (optional).
- BAN logic does not prove correctness of the protocol; but it helps to find subtle errors.

Part V

Secure Communication Protocols

This part illustrates how the cryptographic primitives introduced so far are used to secure communication over the Internet. It is important to recall that the Internet was not designed with security in mind. Almost all early Internet protocols provided no serious security services. Protocol designers started in the early 1990s to introduce security features into existing protocols, sometimes successful, but also often failing to produce a solution that is both secure and easy to adopt and use.

We look at some of the more successful secure protocol designs. People interested in understanding secure protocol designs in more detail should, however, also study the failed designs. There is often quite a bit to learn from design failures.

We will first look at an email security solution (PGP) that can be used in general to protect documents. We then discuss transport layer security (TLS), which is very widely used to secure communication over the Internet. Next, we look at the secure shell (SSH) protocol, which is the protocol of choice for system administration tasks and command line access to remote systems. We finally discuss basic principles of the domain name security solution DNSSEC (even though one can argue that it is not yet clear whether DNSSEC should be counted as a success or a failure).

Pretty Good Privacy

24 Pretty Good Privacy

25 Transport Layer Security

26 Secure Shell

Pretty Good Privacy (PGP)

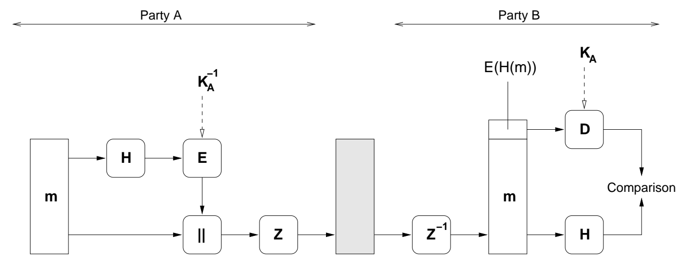
- PGP was developed by Philip Zimmerman in 1991 and it is rather famous because PGP also demonstrated why patent laws and export laws in a globalized world need new interpretations.
- There are nowadays several independent PGP implementations.
- The underlying PGP specification is now called open PGP (RFC 4880).
- A competitor to PGP is S/MIME (which relies on X.509 certificates).

A popular implementation of RFC 4880 [5] is the Gnu Privacy Guard (gpg).

PGP (or GPG) is used to sign software updates in the Debian and Ubuntu Linux distributions. The keys used to sign software release files are distributed using a `debian-archive-keyring` package. The release file contains the checksums of the packages.

Note that using PGP (or GPG) is not always a good idea. For a discussion, see the paper "Off-the-Record Communication, or, Why Not To Use PGP", WPES'04, ACM, Ocober 28 2004.

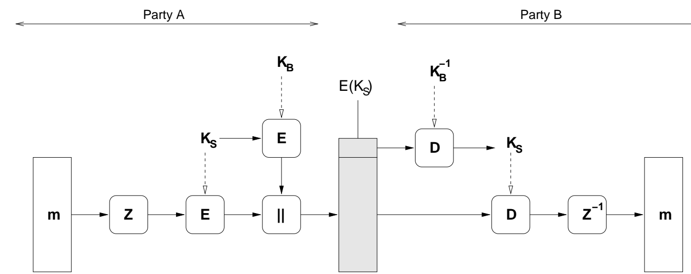
PGP Signatures



- A computes $c = Z(E_{K_A^{-1}}(H(m))||m)$
- B computes $Z^{-1}(c)$, splits the message and checks the signature by computing $D_{K_A}(E_{K_A^{-1}}(H(m)))$ and then checking the hash $H(m)$.

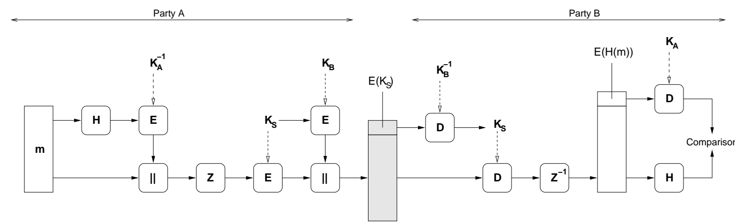
PGP originally used the hash-function MD5, the public-key algorithm RSA and zlib compression. Newer versions support crypto agility.

PGP Confidentiality



- A encrypts the message using the key K_s generated by the sender and appended to the encrypted message.
- The key K_s is protected by encrypting it with the public key K_B.
- Symmetric encryption is fast while public-key algorithms make it easier to exchange keys.

PGP Signatures and Confidentiality



- Signature and confidentiality can be combined as shown above.
- PGP uses in addition Radix-64 encoding (a variant of base-64 encoding) to ensure that messages can be represented using the ASCII character set.
- PGP supports segmentation/reassembly functions for very large messages.

PGP Key Management

- Keys are maintained in so called key rings (one for public keys and one for private keys).
- Key generation utilizes various sources of random information (/dev/random if available) and symmetric encryption algorithms to generate good key material.
- So called "key signing parties" are used to sign keys of others and to establish a "web of trust" in order to avoid centralized certification authorities.

PGP keys need to be signed to build the web of trust. PGP key signing often takes place in so called PGP key signing parties. Here is a short description how this works (using the gpg command line tool):

- Create a gpg key and publish it:

  ```
  gpg --full-generate-key
  ```

  Inspect your keys and get the key identifier of your public key:

  ```
  gpg --list-keys [--fingerprint]
  MYKEYID='...'
  ```

- Send your public key to a key server:

  ```
  gpg --send-key $MYKEYID
  ```

- Prepare for key signing (print out fingerprints of your key)

  ```
  gpg -v --fingerprint $MYKEYID
  ```

- Signing keys of others, identified by their key identifier:

  ```
  YOURKEYID='...'
  gpg --recv-keys $YOURKEYID
  gpg --fingerprint $YOURKEYID
  ```

  Verify the fingerprints and the identity of the person. Then sign the key:

  ```
  gpg --sign-key $YOURKEYID
  ```

  Send the signature back to the owner of the key:

  ```
  gpg --armor --export $YOURKEYID \
    | gpg --encrypt -r $YOURKEYID --armor --output $YOURKEYID-signedby-$MYKEYID.asc
  ```

- Importing signatures and publishing your signed public key:

  ```
  gpg -d $MYKEY-signedBy-$YOURKEYID.asc | gpg --import
  ```

  Send your key with the signatures to a key server:

  ```
  gpg --send-key $MYKEYID
  ```

PGP Private Key Ring

Timestamp	Key ID	Public Key	Encrypted Private Key	User ID
⋮	⋮	⋮	⋮	⋮
T_i	$K_i \bmod 2^{64}$	K_i	$E_{H(P_i)}(K_i^{-1})$	User_i
⋮	⋮	⋮	⋮	⋮

- Private keys are encrypted using $E_{H(P_i)}()$, which is a symmetric encryption function using a key which is derived from a hash value computed over a user supplied passphrase P_i.
- The Key ID is taken from the last 64 bits of the key K_i.

PGP Public Key Ring

Timestamp	Key ID	Public Key	Owner Trust	User ID	Signatures	Sig. Trust(s)
⋮	⋮	⋮	⋮	⋮	⋮	⋮
T_i	$K_i \bmod 2^{64}$	K_i	otrust_i	User_i	...	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮

- Keys in the public key ring can be signed by multiple parties. Every signature has an associated trust level:
  1. undefined trust
  2. usually not trusted
  3. usually trusted
  4. always trusted
- Computing a trust level for new keys which are signed by others (trusting others when they sign keys).

Transport Layer Security

24 Pretty Good Privacy

25 Transport Layer Security

26 Secure Shell

Transport Layer Security

- Transport Layer Security (TLS), formerly known as Secure Socket Layer (SSL), was created by Netscape to secure data transfers on the Web (i.e., to enable commerce on the Web)
- As a user-space implementation, TLS can be shipped as part of applications (Web browsers) and does not require operating system support
- TLS uses X.509 certificates to authenticate servers and clients (although TLS layer client authentication is not often used)
- TLS is widely used to secure application protocols running over TCP (e.g., http, smtp, ftp, telnet, imap, ...)
- A datagram version of TLS called DTLS can be used with protocols running over UDP

TLS 1.2 is defined in RFC 5246 [8] and TLS 1.3 has been published recently in RFC 8446 [26]. DTLS 1.2 is defined in RFC 6347 [27]. A good article describing the design of DTLS is [21].

History of TLS and SSL

Name	Organization	Published	Wire Version
SSL 1.0	Netscape	unpublished	1.0
SSL 2.0	Netscape	1995	2.0
SSL 3.0	Netscape	1996	3.0
TLS 1.0	IETF	1999	3.1
TLS 1.1	IETF	2006	3.2
TLS 1.2	IETF	2008	3.3
TLS 1.3	IETF	2018	3.3 + supported_versions

- TLS 1.3 is brand new, this material follows TLS 1.2 and TLS 1.3

Attacks on TLS 1.2 have been increasing in recent years. TLS 1.3 introduces a radically different handshake protocol and it removes a large collection of problematic constructs. However, only future will tell whether TLS 1.3 is robust to attacks.

TLS Protocols

- The *Handshake Protocol* authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material.
- The *Alert Protocol* communicates alerts such as closure alerts and error alerts.
- The *Record Protocol* uses the parameters established by the handshake protocol to protect traffic between the communicating peers.
- The Record Protocol is the lowest internal layer of TLS and it carries the handshake and alert protocol messages as well as application data.

TLS Record Protocol

Record Protocol

The record protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, adds a message authentication code, and encrypts and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients.

- The record layer is used by the handshake protocol, the change cipher spec protocol (only TLS 1.2), the alert protocol, and the application data protocol.
- The fragmentation and reassembly provided does not preserve application message boundaries.

TLS defines message formats using a notation that resembles C. Here is the definition of the record protocol of TLS 1.2. Note that tha record can be either `TLSPlaintext`, `TLSCompressed`, or `TLSCiphertext`, where the `TLSCiphertext` supports multiple cipher types.

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    ContentType type;       /* same as TLSPlaintext.type */
    ProtocolVersion version;/* same as TLSPlaintext.version */
    uint16 length;
    opaque fragment[TLSCompressed.length];
} TLSCompressed;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (SecurityParameters.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
        case aead:   GenericAEADCipher;
    } fragment;
} TLSCiphertext;
```

TLS 1.3 drops `TLSCompressed` and simplifies `TLSCiphertext` by always using ciphers modeled as Authenticated Encryption with Additional Data (AEAD).

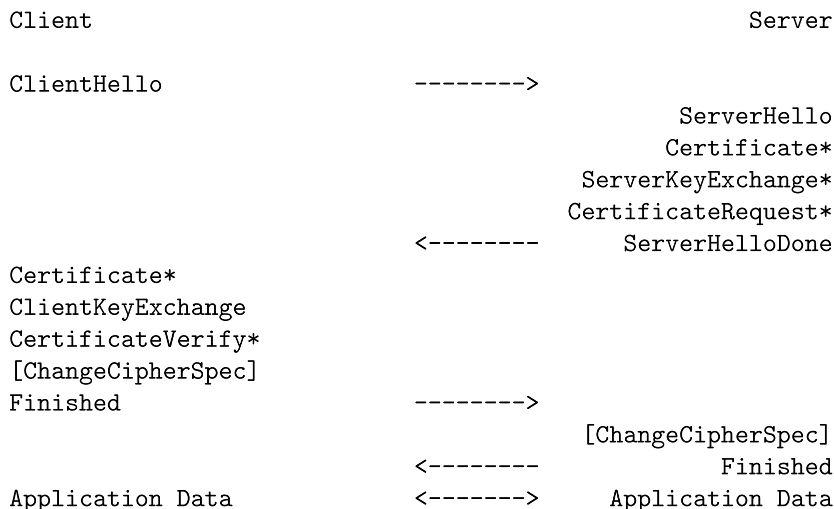
```
uint16 ProtocolVersion;

enum {
    invalid(0), change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

TLS Handshake Protocol

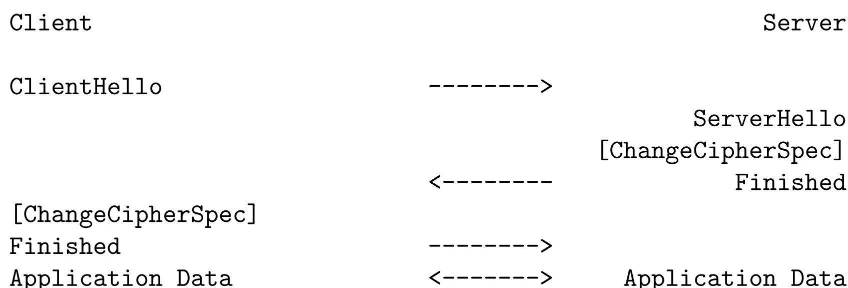
Handshake Protocol

- Exchange messages to agree on algorithms, exchange random numbers, and check for session resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and the exchanged random numbers.
- Provide security parameters to the record layer.
- Allow client and server to verify that the peer has calculated the same security parameters and that the handshake completed without tampering by an attacker.

A full TLS 1.2 handshake is used to establish a session key. Client authentication using a certificate is supported but not mandatory to use. A full TLS 1.2 handshake requires two round-trips before application data can be sent.

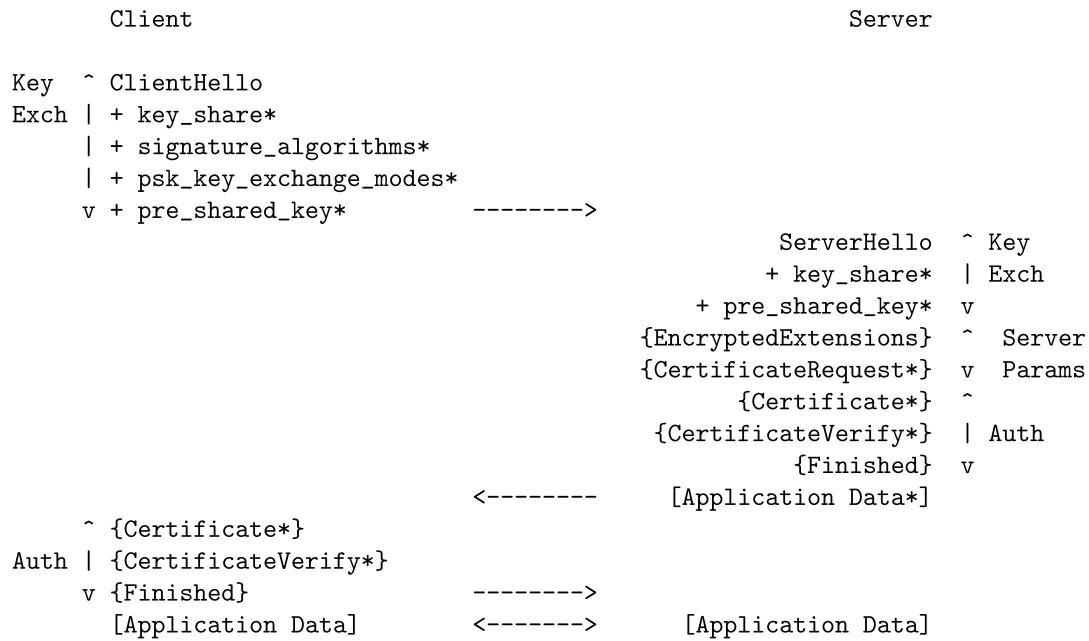


Full handshakes are expensive. A session resumption mechanism was designed for TLS 1.2 to improve performance in situations where sessions are short and frequent.

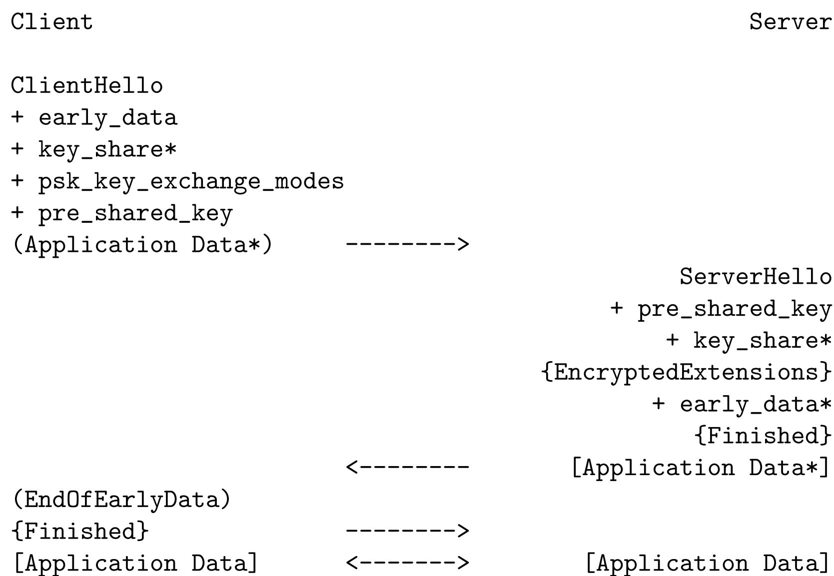


TLS 1.2 session resumption requires only one round-trip and it saves CPU intensive asymmetric crypto operations.

TLS 1.3 has very different handshake exchanges. The full TLS 1.3 handshake looks like this:



TLS 1.3 supports a so-called 0-rtt (zero round-trip) mode:



Note that early data enjoys less cryptographic strong protection.

TLS Change Cipher Spec Protocol

Change Cipher Spec Protocol

The change cipher spec protocol is used to signal transitions in ciphering strategies.

- The protocol consists of a single ChangeCipherSpec message.
- This message is sent by both the client and the server to notify the receiving party that subsequent records will be protected under the newly negotiated CipherSpec and keys.
- This protocol does not exist anymore in TLS 1.3.

TLS Alert Protocol

Alert Protocol

The alert protocol is used to signal exceptions (warnings, errors) that occured during the processing of TLS protocol messages.

- The alert protocol is used to properly close a TLS connection by exchanging `close_notify` alert messages.
- The closure exchange allows to detect truncation attacks.

Secure Shell

24 Pretty Good Privacy

25 Transport Layer Security

26 Secure Shell

Secure Shell (SSH)

- SSH provides a secure connection through which user authentication and several inner protocols can be run.
- The general architecture of SSH is defined in RFC 4251.
- SSH was initially developed by Tatu Ylonen at the Helsinki University of Technology in 1995, who later founded SSH Communications Security.
- SSH was quickly adopted as a replacement for insecure remote login protocols such as telnet or rlogin/rsh.
- Several commercial and open source implementations are available running on almost all platforms.
- SSH is a Proposed Standard protocol of the IETF since 2006.

SSH Protocol Layers

1. The **Transport Layer Protocol** provides server authentication, confidentiality, and integrity with perfect forward secrecy
2. The **User Authentication Protocol** authenticates the client-side user to the server
3. The **Connection Protocol** multiplexes the encrypted data stream into several logical channels

⇒ SSH authentication is not symmetric!

⇒ The SSH protocol is designed for clarity, not necessarily for efficiency (shows its academic roots)

The SSH protocol architecture is defined in RFC 4251 [35]. The SSH transport protocol is defined in RFC 4253 [36] and the user authentication protocol in RFC 4252 [33]. RFC 4254 [34] defines the connection protocol.

SSH Keys, Passwords, and Passphrases

Host Key

Every machine must have a public/private host key pair. Host Keys are often identified by their fingerprint.

User Key

Users may have their own public/private key pairs.

User Password

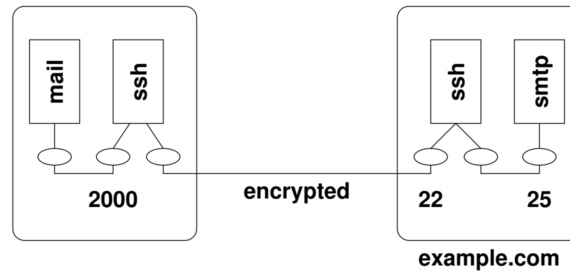
Accounts may have passwords to authenticate users.

Passphrase

The storage of a user's private key may be protected by a passphrase.

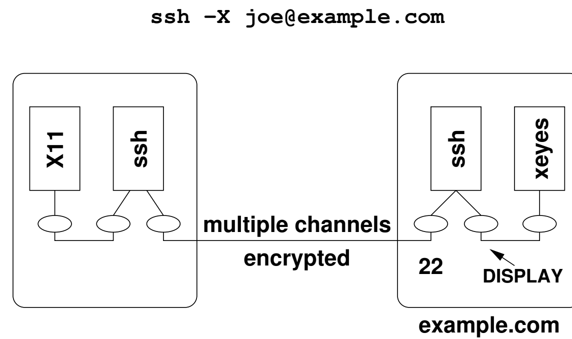
SSH Features: TCP Forwarding

```
ssh -f joe@example.com -L 2000:example.com:25 -N
```



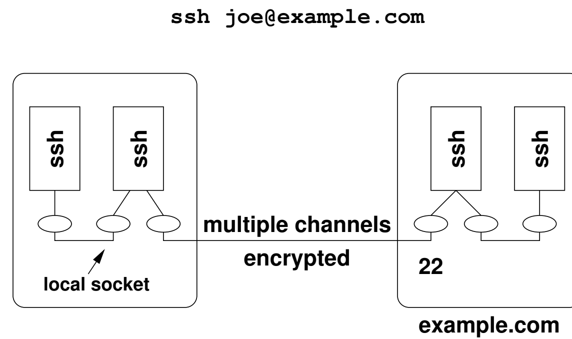
- TCP forwarding allows users to tunnel unencrypted traffic through an encrypted SSH connection.

SSH Features: X11 Forwarding



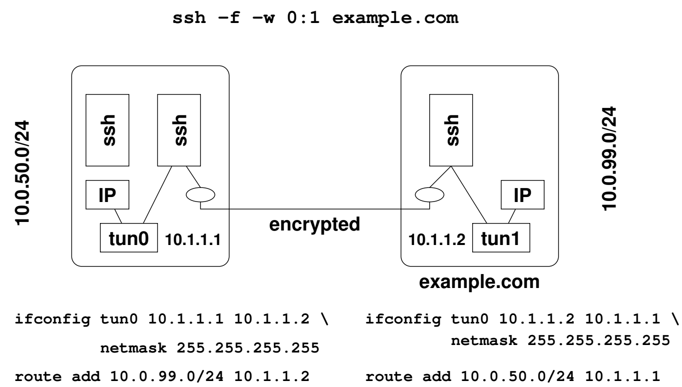
- X11 forwarding is a special application of TCP forwarding allowing X11 clients on remote machines to access the local X11 server (managing the display and the keyboard/mouse).

SSH Features: Connection Sharing



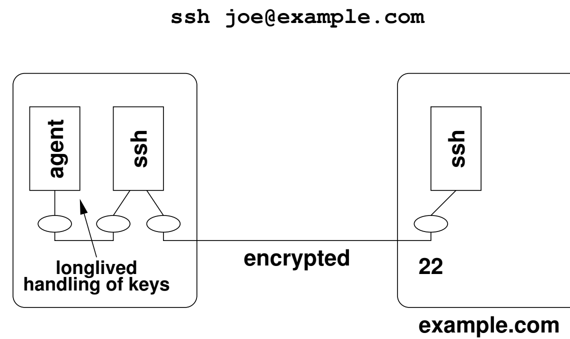
- New SSH connections hook as a new channel into an existing SSH connection, reducing session startup times (speeding up shell features such as tab expansion).

SSH Features: IP Tunneling



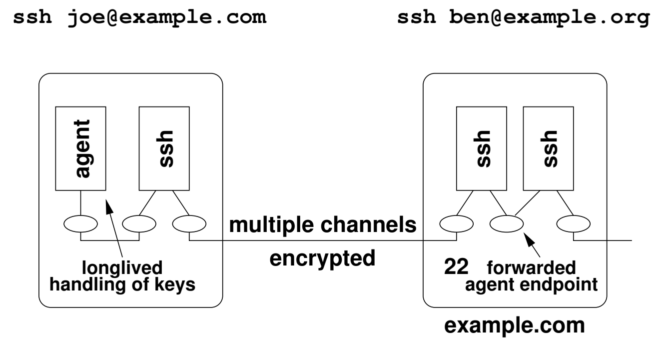
- Tunnel IP packets over an SSH connection by inserting tunnel interfaces into the kernels and by configuring IP forwarding.

SSH Features: SSH Agent



- Maintains client credentials during a login session so that credentials can be reused by different SSH invocations without further user interaction.

SSH Features: SSH Agent Forwarding



- An SSH server emulates an SSH Agent and forwards requests to the SSH Agent of its client, creating a chain of SSH Agent delegations.

SSH Transport Protocol

- Transport Protocol (RFC 4253) provides
  - strong encryption,
  - server authentication,
  - integrity protection, and
  - optionally compression.
- SSH transport protocol typically runs over TCP
- 3DES (required), AES128 (recommended)
- hmac-sha2-256 (recommended, see RFC 6668)
- Automatic key re-exchange, usually after 1 GB of data have been transferred or after 1 hour has passed, whichever is sooner.

SSH Key Exchange

- The SSH host key exchange identifies a server by its hostname or IP address and possibly port number.
- Other key exchange mechanisms use different naming schemes for a host.
- Different key exchange algorithms
  - Diffie-Hellman style key exchange
  - GSS-API style key exchange
- Different Host key algorithms
  - Host key used to authenticate key exchange
  - SSH RSA and DSA keys
  - X.509 (under development)

SSH User Authentication

- Executes after transport protocol initialization (key exchange) to authenticate client.
- Authentication methods:
  - Password (classic password authentication)
  - Interactive (challenge response authentication)
  - Host-based (uses host key for user authentication)
  - Public key (usually DSA or RSA keypairs)
  - GSS-API (Kerberos / NETLM authentication)
  - X.509 (under development)
- Authentication is client-driven.

SSH Connection Protocol

- Allows opening of multiple independent channels.
- Channels may be multiplexed in a single SSH connection.
- Channel requests are used to relay out-of-band channel specific data (e.g., window resizing information).
- Channels commonly used for TCP forwarding.

OpenSSH Privilege Separation

- Privilege separation is a technique in which a program is divided into parts which are limited to the specific privileges they require in order to perform a specific task.
- OpenSSH is using two processes: one running with special privileges and one running under normal user privileges
- The process with special privileges carries out all operations requiring special permissions.
- The process with normal user privileges performs the bulk of the computation not requiring special rights.
- Bugs in the code running with normal user privileges do not give special access rights to an attacker.

Part VI

Information Hiding and Privacy

Cryptographic mechanism can protect information. By encrypting data, only parties with access to the appropriate keys can read or modify the data. There are, however, situations where it is desirable to hide the fact that data exists. Information hiding is a research domain that covers a wide spectrum of methods that are used to make (secret) data difficult to notice [31].

We will first introduce techniques to hide data in other data (steganography) and techniques to proof that a certain data object has a certain origin (watermarks). Afterwards, we will discuss hidden communication channels (covert channels).

We then focus our attention on anonymity. We start by introducing basic terminology (anonymity, unlinkability, undetectability, pseudonymity, identifiability) and then look at the basic principles of mixing networks and onion routing networks.

Steganography and Watermarks

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

For a good introduction into stegangraphy, see the paper by Niels Provos and Peter Honeyman [24].

Information Hiding

Definition (information hiding)

Information hiding aims at concealing the very existence of some kind of information for some specific purpose.

- Information hiding itself does not aim at protecting message content
- Encryption protects message content but by itself does not hide the existence of a message
- Information hiding techniques are often used together with encryption in order to both hide the existence of messages and to protect messages in case their existence is uncovered

Some applications of information hiding:

- Improve confidentiality by hiding the very existence of messages
- Prove ownership of digital media (watermarking)
- Fingerprinting media for tracking purposes
- Hiding communication (covert channels)
- Identification of devices (e.g., printers) used to produce an artefact (e.g., a printed photo)
- Enabling forensics
- ...

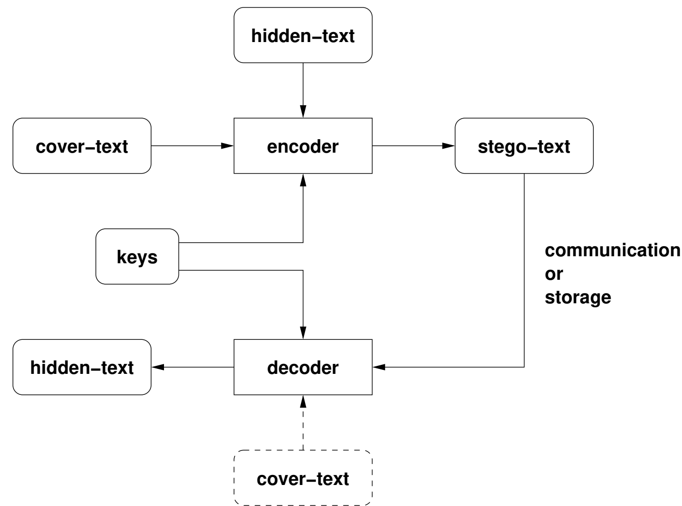
Steganography

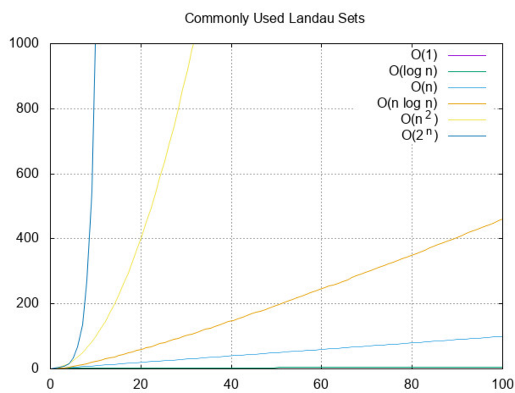
Definition (steganography)

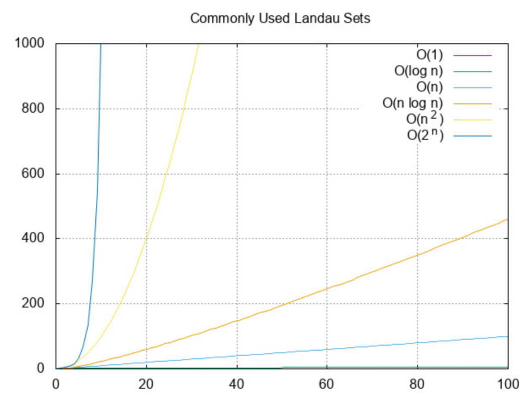
Steganography is the embedding of some information (hidden-text) within digital media (cover-text) so that the resulting digital media (stego-text) looks unchanged (imperceptible) to a human/machine.

- Information hiding explores the fact that there are often unused or redundant bits in digital media that can be used to carry hidden digital information.
- The challenge is to identify unused or redundant bits and to encode hidden digital information in them in such a way that the existence of hidden information is difficult to observe.

Steganography Workflow








```
$ gnuplot landau.gp > landau.jpg
$ steghide embed -cf landau.jpg -ef landau.gp -sf landau.jpeg
$ steghide extract -sf landau.jpeg -xf -
set term jpeg
set title "Commonly Used Landau Sets"
set grid
set xrange [0:100]
set yrange [0:1000]

plot 1 title "O(1)", \
     log(x) title "O(log n)", \
     x title "O(n)", \
     x*log(x) title "O(n log n)", \
     x**2 title "O(n^2)", \
     2**x title "O(2^n)"
```

Types of Cover Media

- Information can be hidden in various cover media types:
  - Image files
  - Audio files
  - Video files
  - Text files
  - Software (e.g., executable files, source code)
  - Network traffic (e.g., covert channels)
  - Storage devices (e.g., steganographic file systems)
  - Events (e.g., timing covert channels, signaling covert channels)
  - ...
- Media types of large size usually make it easier to hide information.
- Robust steganographic methods may survive some typical modifications of stego-texts (e.g., cropping or recoding of images).

Watermarking

Definition (watermarking)

Watermarking is the embedding some information (watermark) within a digital media (cover-text) so that the resulting digital media looks unchanged (imperceptible) to a human/machine.

- Watermarking:
  - The hidden information itself is not important.
  - The watermark says something about the cover-text.
- Steganography:
  - The cover-text is not important, it only conveys the hidden information.
  - The hidden text is the valuable information, and it is independent of cover-text.

Digital watermarks are widely used to for copyright protection and source tracking purposes.

Some modern laser printers add tiny yellow dots to each page. The barely-visible dots contain encoded printer serial numbers and date and time stamps.

Compared to steganography algorithms, watermark algorithms usually only need to store small amounts of data. Watermarking algorithms are usually designed to produce robust watermarks (watermarks survive transformations applied to the cover text) and to be difficult to detect and to make it hard to remove watermarks.

One specific application of watermarking is the detection of modifications of digitial media. For example, image processing tools can make significant changes and tampered "fake" images may later be used to support false claims. By embedding a cryptographic hash computed over an image and a key known only to the source of an image as a watermark in an image, it can be possible to detect attempts to edit images.

In the software industry, watermarks may be carried in executable program code in order to track copies and to be able to claim that illegal copies of software originate from a certain customer.

Classification of Steganographic Algorithms

- fragile vs. robust
  - Fragile: Modifiations of stego-text likly destroys hidden text.
  - Robust: Hidden text is likely to survive modifications of the stego-text.
- blind vs. semi-blind vs. non-blind
  - Blind requires the original cover-text for detection / extraction.
  - Semi-blind needs some information from the embedding but not the whole cover-text
  - Non-blind does not need any information for detection / extraction.
- pure vs. symmetric (secret key) vs. asymmetric (public key)
  - Pure needs no key for detection / extraction.
  - Secret key needs a symmetric key for embedding and extraction.
  - Public key needs a secret key for embedding and a public key for extraction.

LSB-based Image Steganography

- Idea:
  - Some image formats encode a pixel using three 8-bit color values (red, green, blue).
  - Changes in the least-significant bits (LSB) are difficult for humans to see.
- Approach:
  - Use a key to select some least-significant bits of an image to embed hidden information.
  - Encode the information multiple times to achieve some robustness against noise.
- Problem:
  - Existence of hidden information may be revealed if the statistical properties of least-significant bits change.
  - Fragile against noise such as compression, resizing, cropping, rotating or simply additive white Gaussian noise.

DCT-based Image Steganography

- Idea:
  - Image formats such as JPEG use discrete cosine transforms (DCT) to encode image data.
  - The manipulation happens in the frequency domain instead of the spatial domain and this reduces visual attacks against the JPEG image format.
- Approach:
  - Replace the least-significant bits of some of the discrete cosine transform coefficients.
  - Use a key to select some DCT coefficients of an image to embed hidden information.
- Problem:
  - Existence of hidden information may be revealed if the statistical properties of the DCT coefficients are changed.
  - This risk may be reduced by using an pseudo-random number generator to select coefficients.

Covert Channels

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

For an overview of covert channel, see the survery paper by Steffen Wendzel et al. [32].

Covert Channels

- Covert channels represent unforeseen communication methods that break security policies. Network covert channels transfer information through networks in ways that hide the fact that communication takes place (hidden information transfer).
- Covert channels embed information in
  - header fields of protocol data units (protocol messages)
  - the timing of protocol data units (e.g., inter-arrival times)
- We are not considering here covert channels that are constructed by exchanging steganographic objects in application messages.

Covert Channel Patterns

P1 Size Modulation Pattern
The covert channel uses the size of a header field or of a protocol message to encode hidden information.

P2 Sequence Pattern
The covert channel alters the sequence of header fields to encode hidden information.

P3 Add Redundancy Pattern
The covert channel creates new space within a given header field or within a message to carry hidden information.

P4 PDU Corruption/Loss Pattern
The covert channel generates corrupted protocol messages that contain hidden data or it actively utilizes packet loss to signal hidden information.

Covert Channel Patterns

P5 Random Value Pattern
The covert channel embeds hidden data in a header field containing a "random" value.

P6 Value Modulation Pattern
The covert channel selects one of values a header field can contain to encode a hidden message.

P7 Reserved/Unused Pattern
The covert channel encodes hidden data into a reserved or unused header field.

P8 Inter-arrival Time Pattern
The covert channel alters timing intervals between protocol messages (inter-arrival times) to encode hidden data.

Covert Channel Patterns

P9 Rate Pattern
The covert channel sender alters the data rate of a traffic flow from itself or a third party to the covert channel receiver.

P10 Protocol Message Order Pattern
The covert channel encodes data using a synthetic protocol message order for a given number of protocol messages flowing between covert sender and receiver.

P11 Re-Transmission Pattern
A covert channel re-transmits previously sent or received protocol messages.

Anonymization Terminology

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

This section is based on: Andreas Pfitzmann, Marit Hansen: A Proposal for Talking about Privacy by Data Minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management, Aug. 10, 2010 (v.34)

https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf

Anonymity

Definition (anonymity)

Anonymity of a subject from an attacker's perspective means that the attacker cannot sufficiently identify the subject within a set of subjects, the anonymity set.

- All other things being equal, anonymity is the stronger, the larger the respective anonymity set is and the more evenly distributed the sending or receiving, respectively, of the subjects within that set is.
- Robustness of anonymity characterizes how stable the quantity of anonymity is against changes in the particular setting, e.g., a stronger attacker or different probability distributions.

Unlinkability and Linkability

Definition (unlinkability)

Unlinkability of two or more items of interest (IOIs) (e.g., subjects, messages, actions, ...) from an attacker's perspective means that within the system, the attacker cannot sufficiently distinguish whether these IOIs are related or not.

Definition (linkability)

Linkability of two or more items of interest (IOIs) (e.g., subjects, messages, actions, ...) from an attacker's perspective means that within the system, the attacker can sufficiently distinguish whether these IOIs are related or not.

Anonymity can be expressed in terms of unlinkability:

- Sender anonymity of a subject means that to this potentially sending subject, each message is unlinkable.
- Recipient anonymity of a subject means that to this potentially receiving subject, each message is unlinkable.
- Relationship anonymity of a pair of subjects, the potentially sending subject and the potentially receiving subject, means that to this potentially communicating pair of subjects, each message is unlinkable.

Undetectability and Unobservability

Definition (undetectability)

Undetectability of an item of interest (IOI) from an attacker's perspective means that the attacker cannot sufficiently distinguish whether it exists or not.

Definition (unobservability)

Unobservability of an item of interest (IOI) means

- undetectability of the IOI against all subjects uninvolved in it and
- anonymity of the subject(s) involved in the IOI even against the other subject(s) involved in that IOI.

- Sender unobservability then means that it is sufficiently undetectable whether any sender within the unobservability set sends. Sender unobservability is perfect if and only if it is completely undetectable whether any sender within the unobservability set sends.
- Recipient unobservability then means that it is sufficiently undetectable whether any recipient within the unobservability set receives. Recipient unobservability is perfect if and only if it is completely undetectable whether any recipient within the unobservability set receives.
- Relationship unobservability then means that it is sufficiently undetectable whether anything is sent out of a set of could-be senders to a set of could-be recipients.

Relationships

With respect to the same attacker, the following relationships hold:

- unobservability ⇒ anonymity
- sender unobservability ⇒ sender anonymity
- recipient unobservability ⇒ recipient anonymity
- relationship unobservability ⇒ relationship anonymity

We also have:

- sender anonymity ⇒ relationship anonymity
- recipient anonymity ⇒ relationship anonymity
- sender unobservability ⇒ relationship unobservability
- recipient unobservability ⇒ relationship unobservability

The usual concept to achieve undetectability of IOIs at some layer, e.g., sending meaningful messages, is to achieve statistical independence of all discernible phenomena at some lower implementation layer. An example is sending dummy messages at some lower layer to achieve e.g., a constant rate flow of messages looking, by means of encryption, randomly for all parties except the sender and the recipient(s).

Pseudonymity

Definition (pseudonym)

A *pseudonym* is an identifier of a subject other than one of the subject's real names. The subject, which the pseudonym refers to, is the holder of the pseudonym.

Definition (pseudonymity)

A subject is *pseudonymous* if a pseudonym is used as identifier instead of one of its real names. *Pseudonymity* is the use of pseudonyms as identifiers.

- Sender pseudonymity is defined as the sender being pseudonymous, recipient pseudonymity is defined as the recipient being pseudonymous.
- A digital pseudonym can be realized as a public key to test digital signatures where the holder of the pseudonym can prove holdership by forming a digital signature, which is created using the corresponding private key. An example would be PGP keys.
- A public key certificate bears a digital signature of a so-called certification authority and provides some assurance to the binding of a public key to another pseudonym, usually held by the same subject. In case that pseudonym is the civil identity (the real name) of a subject, such a certificate is called an identity certificate.

Identifiability and Identity

Definition (identifiability)

Identifiability of a subject from an attacker's perspective means that the attacker can sufficiently identify the subject within a set of subjects, the identifiability set.

Definition (identity)

An identity is any subset of attribute values of an individual person which sufficiently identifies this individual person within any set of persons. So usually there is no such thing as "the identity", but several of them.

- Identity can be explained, from a psychological perspective, as an exclusive perception of life, integration into a social group, and continuity, which is bound to a body and – at least to some degree – shaped by society.
- Identity can be explained and defined, from a more mathematical perspective, as a property of an entity in terms of the opposite of anonymity and the opposite of unlinkability.
- Identity enables both to be identifiable as well as to link IOIs because of some continuity of life.

Identity Management

Definition (identity management)

Identity management means managing various partial identities (usually denoted by pseudonyms) of an individual person, i.e., administration of identity attributes including the development and choice of the partial identity and pseudonym to be (re-)used in a specific context or role.

- A partial identity is a subset of attribute values of a complete identity, where a complete identity is the union of all attribute values of all identities of this person.
- A pseudonym might be an identifier for a partial identity.

Given the restrictions of a set of applications, identity management is called privacy-enhancing if it sufficiently preserves unlinkability (as seen by an attacker) between the partial identities of an individual person required by the applications.

Mixes and Onion Routing

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

Mix Networks

- A mix network uses special proxies called mixes to send date from a source to a destination.
- The mixes filter, collect, recode, and reorder messages in order to hide conversations. Basic operations of a mix:
  1. Removal of duplicate messages (an attacker may inject duplicate message to infer something about a mix.
  2. Collection of messages in order to create an ideally large anonymity set.
  3. Recoding of messages so that incoming and outgoing messages cannot be linked.
  4. Reordering of messages so that order information cannot be used to link incoming and outgoing messages.
  5. Padding of messages so that message sizes do not reveal information to link incoming and outgoing messages.

Mix networks were introduced in 1981 as a technique to provide anonymous email delivery [7]. Mix networks get their security from the mixing done by their component mixes, and may or may not use route unpredictability to enhance security.

Notation:

A, B	principals
$M_1, M_2, \ldots$	mixes
K_X	public key of X
K_X^{-1}	private key of X
k_i	ephemeral symmetric keys
N_i	nonces
m	message

- Sender anonymity: ($A \to M_1 \to M_2 \to B$)

$$A \to M_1 : \{N_1, M_2, \{N_2, B, \{m\}_{K_B}\}_{K_{M_2}}\}_{K_{M_1}} \qquad M_1 \text{ extracts } M_2$$
$$M_1 \to M_2 : \{N_2, B, \{m\}_{K_B}\}_{K_{M_2}} \qquad M_2 \text{ extracts } B$$
$$M_2 \to B : \{m\}_{K_B} \qquad B \text{ extracts the message } m$$

- Receiver anonymity: ($A \to M_1 \to M_2 \to B$)

  The receiver B chooses a set a of mixes and for every mix an ephemeral symmetric key k_i. The receiver then generates a return address R:

$$R = \{k_0, M_1, \{k_1, M_2, \{k_2, B\}_{K_{M_2}}\}_{K_{M_1}}\}_{K_A}$$

  The return address is sent to A as described above:

$$B \to M_2 : \{N_2, M_1, \{N_1, A, \{R\}_{K_A}\}_{K_{M_1}}\}_{K_{M_2}} \qquad M_2 \text{ extracts } M_1$$
$$M_2 \to M_1 : \{N_1, A, \{R\}_{K_A}\}_{K_{M_1}} \qquad M_1 \text{ extracts } A$$
$$M_1 \to A : \{R\}_{K_A} \qquad A \text{ extracts the return address } R$$

  The sender A extracts k_0 from R and sends the following to M_1:

$$A \to M_1 : \{m\}_{k_0}, \{k_1, M_2, \{k_2, B\}_{K_{M_2}}\}_{K_{M_1}} \qquad M_1 \text{ extracts } k_1 \text{ and } M_2$$
$$M_1 \to M_2 : \{\{m\}_{k_0}\}_{k_1}, \{k_2, B\}_{K_{M_2}} \qquad M_2 \text{ extracts } k_2 \text{ and } B$$
$$M_2 \to B : \{\{\{m\}_{k_0}\}_{k_1}\}_{k_2} \qquad B \text{ extracts the message } m$$

Onion Routing

- A message m it sent from the source S to the destination T via an overlay network consisting of the intermediate routers R_1, R_2, ..., R_n, called a circuit.
- A message is cryptographically wrapped multiple times such that every router R unwraps one layer and thereby learns to which router the message needs to be forwarded next.
- To preserve the anonymity of the sender, no node in the circuit is able to tell whether the node before it is the originator or another intermediary like itself.
- Likewise, no node in the circuit is able to tell how many other nodes are in the circuit and only the final node, the "exit node", is able to determine its own location in the chain.

Onion routing systems primarily get their security from choosing routes that are difficult for the adversary to observe.

Tor

- Tor is an anonymization network operated by volunteers supporting the Tor project.
- Every Tor router has a long-term identity key and a short-term onion key.
- The identity key is used to sign TLS certificates and the onion key is used to decrypt messages to setup circuits and ephemeral keys.
- TLS is used to protect communication between onion routers.
- Directory servers provide access to signed state information provided by Tor routers.
- Applications build circuits based on information provided by directory servers.

A first working version of Tor was announced in 2002. The Tor project receive initially funding from US government organizations such as the Defense Advanced Research Projects Agency (DARPA). Since 2006, the Tor projekt is supported by The Tor Project, Inc, a non-profit organization. The Tor project web site is at `https://www.torproject.org/`. A technical description of the second version of Tor can be found in [10].

Tor aims at protecting the traffic in transit, it is of limited help if application protocols running over Tor circuits leak information that allows to link traffic to identities. Since Tor does aim at supporting interactive applications, it is in general subject to traffic analysis attacks and in particular timing analysis (where traffic and server traces are linked based on timing properties).

Due to the Tor design, exit nodes get access to the original messages. Hence, in order to be protected against compromised exit nodes, it is still crucial to use end-to-end encryption with Tor.

Part VII

Operating System Security

Authentication, Authorization, Auditing, Isolation

- Authentication
  - Who is requesting an action?
- Authorization
  - Is a principal allowed to execute an action on this object?
- Auditing
  - Record evidence for decision being made in an audit-trail.
- Isolation
  - Isolate system components from each other to create sandboxes

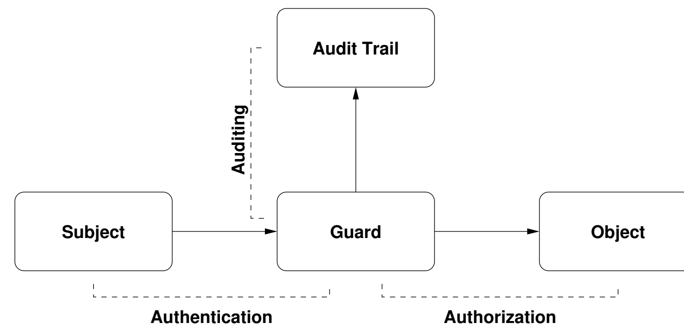
Basic authentication at the operating system level is typically implemented using passwords, which is known to be problematic. On mobile devices, we meanwhile often find in addition biometric authentication mechanisms. Operating system access over the network is often using assymmetric cryptographic key mechanisms. In Unix-like systems, the authentication resolves to a user identifier (uid) of processes and the kernel makes an important distinction between the user identifier (uid) and the effective user identifier (euid).

Authorization answers the question which operations are allowed against an object. This question is answered using an authorization (or access control) policy. The specification of authorization policies is complex and there are different approaches to specify authorization policies.

Auditing is used to keep a log (an audit trail) of the decisions made. This is essential for debugging purposes but also for forensics in case a system was attacked or information was leaked to principals who should not have had access to the information. A good audit trail is extremely important but also highly sensitive information.

Isolation is means to control complexity. Operating systems typically isolate the memory used by processes from each other. By isolating processes into containers, it is possible to prevent misbehaving processes from damaging the entire system. Trusted computing platforms are moving towards hardware-assisted isolation of critical software components.

Lampson Model



- This basic model works well for modeling static access control systems.
- Dynamic access control systems allowing dynamic changes to the access control policy are difficult to model with this approach.

The original paper by Lampson [18] uses a slightly different terminology.

Authentication

31 Access Control Principles

Access Control

31 Access Control Principles

Subjects, Objects, Rights

- Subjects (S): set of active objects
  - processes, users, ...
- Objects (O): set of protected entities
  - files, directories, ...
  - memory, devices, sockets, ...
  - processes, memory, ...
- Rights (R): set of operations a subject can perform on an object
  - create, read, write, delete ...
  - execute ...

Lampson's Access Control Matrix

- Subjects $s_i \in S$ are row headings.
- Objects $o_j \in O$ are column headings.
- The rights of subject s_i when accessing object o_j are defined by the entry $M[s_i, o_j]$ of the access control matrix M
- Another way to look at access control rights is that the rights $r \in R$ are a function mapping $(S \times O) \rightarrow R^*$
- Since the access control matrix can be huge, it is necessary to find ways to express it in a format that is lowering the cost for maintaining it.

An access control matrix is great in theory by difficult in practice since the product of all subjects against all objects is huge. Hence, it is necessary to find representations that reduce the size of the access control matrix and which make the management of access rights feasible for a security administrator. Two widely used approaches are access control lists and capabilities.

Discretionary vs. Mandatory Access Control

- Discretionary Access Control (DAC)
  - Subjects (with certain permissions, i.e., ownership) can define access control rules to allow or deny access to an object
  - It is the at the subject's discretionary what to allow to whom.
- Mandatory Access Control (MAC)
  - System mechanisms control access to an object and an individual subject cannot alter the access rights.
  - What is allowed is mandated by the system (or the security administrator of the system).

Unix filesystem permissions are an example of discretionary access control. The owner of a file controls who is allowed to access the file in which way.

Mandatory access control is frequently used by security critical systems to enforce access control rules. Early forms of mandatory access control were often using multi-level security systems, where objects are classified into security levels and subjects are allowed access to objects in the security security level associated with the subject.

Role-based access control models try to simplify the management of access control rules. The basic idea is that subjects are first mapped into roles and access control rules are defined for certain roles. For example, access rights for certain documents may be given to the role of a study program chair instead of specific persons. This has the benefit that the person taking the role of a study program chair can be easily replaced without having to redefine all access control rules for all documents.

Access Control Lists

- An access control list represents a column of the access control matrix.
- Given a set of subjects S and a set of rights R, an access control list of an object $o \in O$ is a set of tuples of $S \times R^*$.
- Example: Unix file system permissions. The inode of a file (the object) stores the whether a user or a group (the subject) has read/write/execute permissions (the rights).

Access Control List Design Issues

- Who can define and modify ACLs?
- Does the ACL support groups or wildcards?
- How are contradictory ACLs handled?
- Is there support for default ACLs?

Capabilities

- A capability represents a row of the access control matrix.
- Given a set of objects o and a set of rights R, a capability of a subject s is a set of tuples of $O \times R^*$.
- Example: An open Unix file descriptor can be seen as a capability. Once opened, the file can be used regardless whether the file is deleted or whether access rights are changed. The capability (the open file descriptor) can be transfered to child processes. Note that passing capabilities to child processes is not meaningful for all capabilities.

Capabilities are like tickets that allow a subject to do certain things. Note that is essential that subjects cannot alter their capabilities in an uncontrolled way. Operating systems therefore typically maintain capabilities in kernel space. The file descriptor, for example, is maintained in the kernel and it cannot be changed to refer to different files without the involvement of the kernel.

Capabilities Design Issues

- How are capabilities stored?
- How are capabilities protected?
- Can capabilities be passed on to other subjects?
- Can capabilities be revoked?

Access Control Lists vs. Capabilities

- Both are theoretically equivalent (since both at the end can represent the same access control matrix)
- Capabilities tend to be more efficient if the common question is "Given a subject, what objects can it access and how?".
- Access control lists tend to be more efficient if the common question is "Given an object, what subjects can access it and how?".
- Access control lists tend to be more popular because they are more efficient when an authorization decision needs to be made.
- Systems often use a mixture of both approaches.

References

[1] M. Abadi and D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1), January 1999.

[2] L. Hornquist Astrand and T. Yu. Deprecate DES, RC4-HMAC-EXP, and Other Weak Cryptographic Algorithms in Kerberos. RFC 6649, Apple, MIT Kerberos Consortium, July 2012.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[4] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Operating Systems Review*, 23(5):1–13, 1989.

[5] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880, PGP Corporation, IKS GmbH, November 2007.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[7] D.L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Independent, RTFM, August 2008.

[9] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 6:644–654, November 1976.

[10] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proc. of the 13th USENIX Security Symposium*, San Diego, August 2004. USENIX.

[11] M.J. Dworkin, E.B. Barker, J.R. Nechvatal, J. Foti, L.E. Bassham, E. Roback, and J.F. Dray. Specification of the advanced encryption standard (aes). Federal Information Processing Standard (FIPS) Publication 197, National Institute of Standards and Technology (NIST), November 2001.

[12] D. Eastlake and T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, Huawei, AT&T Labs, May 2011.

[13] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[14] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, online edition, May 2015.

[16] A. Kehne, J. Schönwälder, and H. Langendörfer. A Nonce-Based Protocol for Multiple Authentications. *ACM Operating System Review*, 26(4):84–89, October 1992.

[17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[18] B.W. Lampson. Computer Security in the Real World. *IEEE Computer*, 37(6):37–46, June 2004.

[19] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, Cisco Systems, January 2008.

[20] R. Milner. *Communicating and Mobile Systems: The pi-calculus*. Cambridge University Press, May 1999.

[21] N. Modadugu and E. Rescorla. The Design and Implementation of Datagram TLS. In *Proc. Network and Distributed System Security Symposium*, San Diego, February 2004.

[22] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[23] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120, USC-ISI, MIT, July 2005.

[24] N. Provos and P. Honeyman. Hide and Seek: An Introduction to Steganography. *IEEE Security and Privacy*, 1(3), June 2003.

[25] M. Raynal. About logical clocks in distributed systems. *Operating Systems Review*, 26(1):41–48, 1992.

[26] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Mozilla, August 2018.

[27] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 6347, RTFM, Google, January 2012.

[28] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 21(2):9–17, 1981.

[29] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2 edition, 2000.

[30] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[31] S. Wendzel W. Mazurczyk. Information Hiding: Challenges for Forensic Experts. *Communications of the ACM*, 61(1):86–94, January 2018.

[32] S. Wendzel, S. Zander, B. Fechner, and C. Herdin. Pattern-Based Survey and Categorization of Network Covert Channel Techniques. *ACM Computing Surveys*, 47(3), April 2015.

[33] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, SSH Communications Security Corp, Cisco Systems, January 2006.

[34] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254, SSH Communications Security Corp, Cisco Systems, January 2006.

[35] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, SSH Communications Security Corp, Cisco Systems, January 2006.

[36] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, SSH Communications Security Corp, Cisco Systems, January 2006.