Secure and Dependable Systems

Jürgen Schönwälder

August 22, 2020

Abstract

This memo provides annotated slides for the Computer Science module “Secure and Dependable Systems” offered at Jacobs University Bremen.

https://cnds.jacobs-university.de/courses/sads-2020

JACOBS
UNIVERSITY

Contents

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.

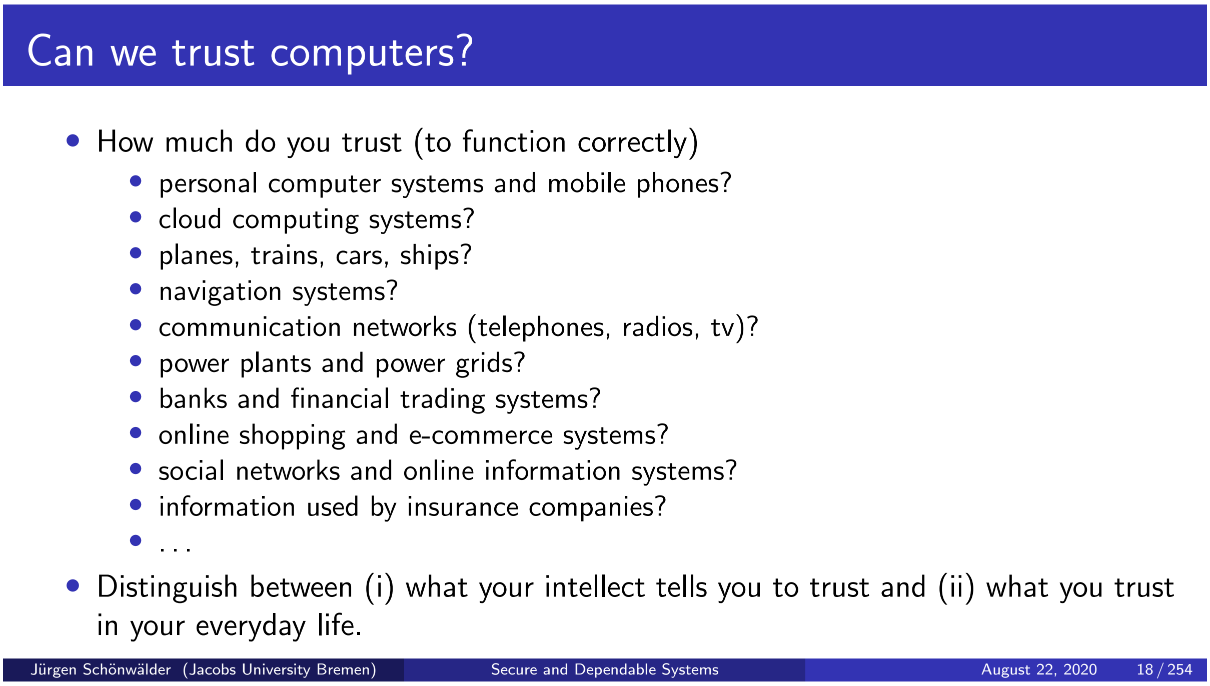
Motivation

4 Motivation

5 Recent 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 [48].

- Stage 1: It is possible to write a program that generates itself. Example:

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

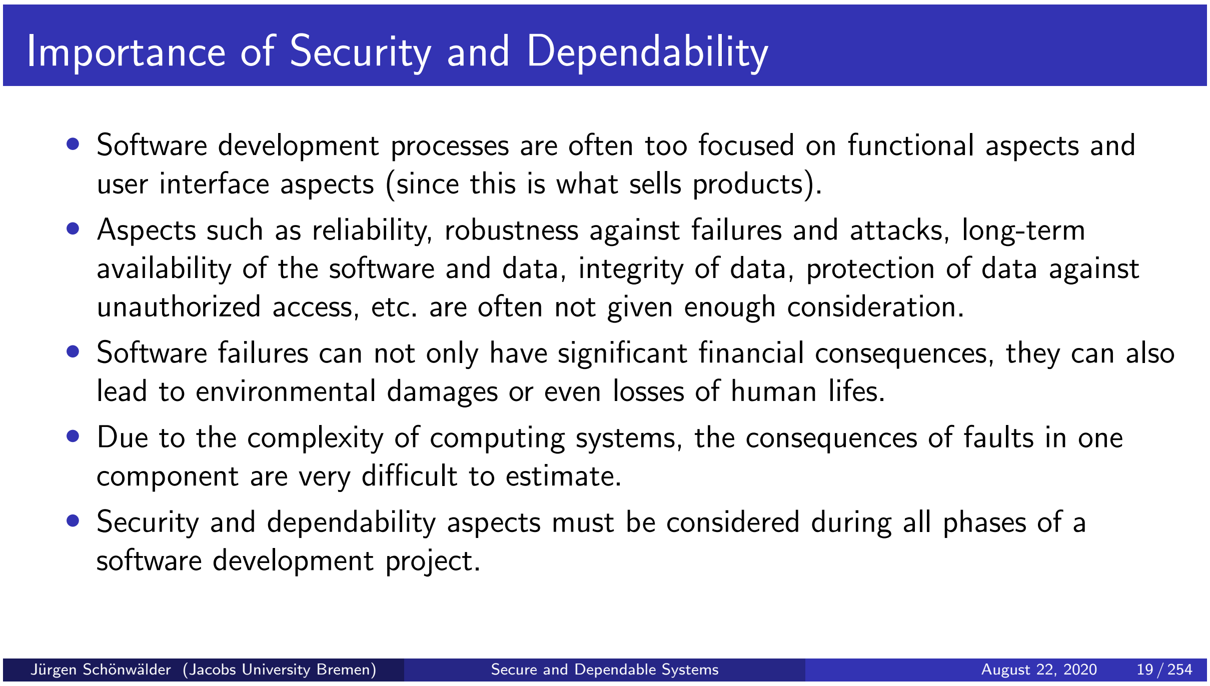
- Stage 2: A compiler can use knowledge already built into the compiler to produce a new compiler. See this example, which may be part of a C compiler implementing the handling of backslash escape sequences.

```
char getchar_escaped(void)
{
    char c;
    if ((c = getchar()) != '\\') return c;
    switch ((c = getchar())) {
    case '\\': return '\\';
    case 'n':  return '\n';
    default:   return c;
    }
}
```

  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 and rely on it.*

In the general case, 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 “dependable” 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 but we can’t go into details of this here. But you are encouraged to lookup up material and to study code of ethics documents specific to computer science. Here is, for example, the list of the *Ten Commandments of Computer Ethics* created in 1992 by the Computer Ethics Institute:

1. Thou shalt not use a computer to harm other people.
2. Thou shalt not interfere with other people’s computer work.
3. Thou shalt not snoop around in other people’s computer files.
4. Thou shalt not use a computer to steal.
5. Thou shalt not use a computer to bear false witness.
6. Thou shalt not copy or use proprietary software for which you have not paid (without permission).
7. Thou shalt not use other people’s computer resources without authorization or proper compensation.
8. Thou shalt not appropriate other people’s intellectual output.
9. Thou shalt think about the social consequences of the program you are writing or the system you are designing.
10. Thou shalt always use a computer in ways that ensure consideration and respect for other humans.

Recent Computing Disasters

4 Motivation

5 Recent Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

IoT Remove Control Light Bulbs 2018

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

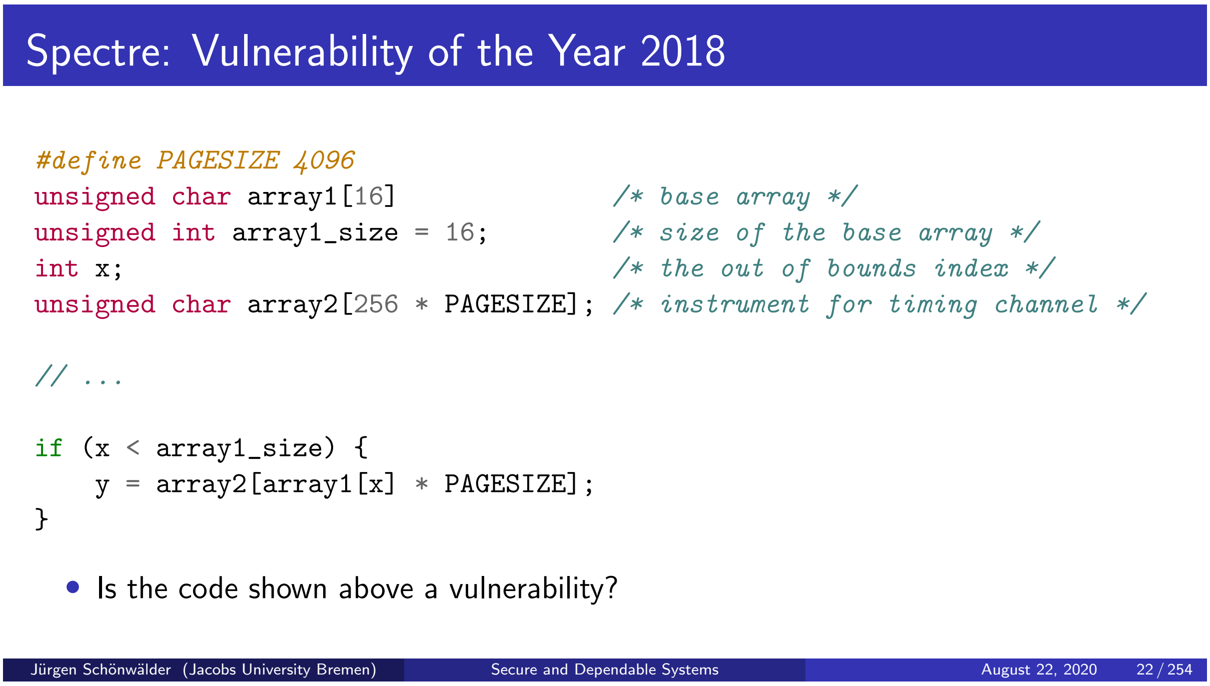
- It stores credentials in plaintext in flash memory that can be read with relatively little effort (even after the bulb has been thrown away).
- It uses a software update mechanism that can be tricked with some minor effort to load compromised firmware (firmware is not signed).
- It leaks unnecessary 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

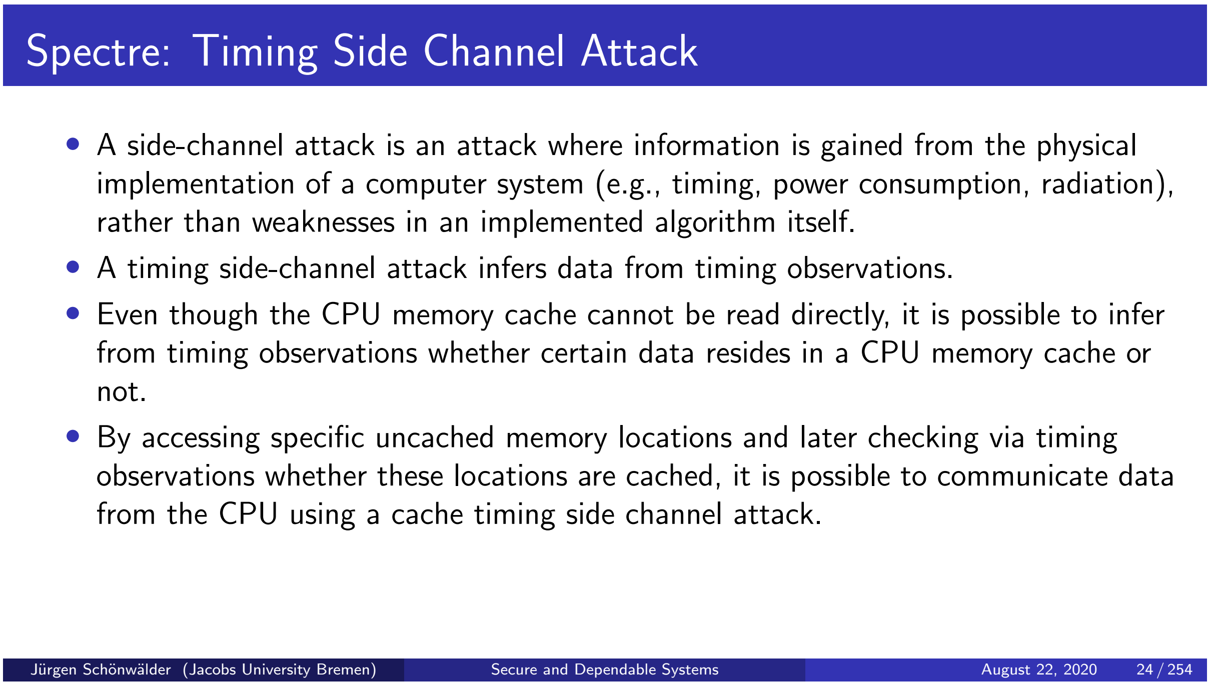
```
#define PAGESIZE 4096
unsigned char array1[16]                /* base array */
unsigned int array1_size = 16;          /* size of the base array */
int x;                                  /* the out of bounds index */
unsigned char array2[256 * PAGESIZE]; /* instrument for timing channel */

// ...

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

- Is the code shown above 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...

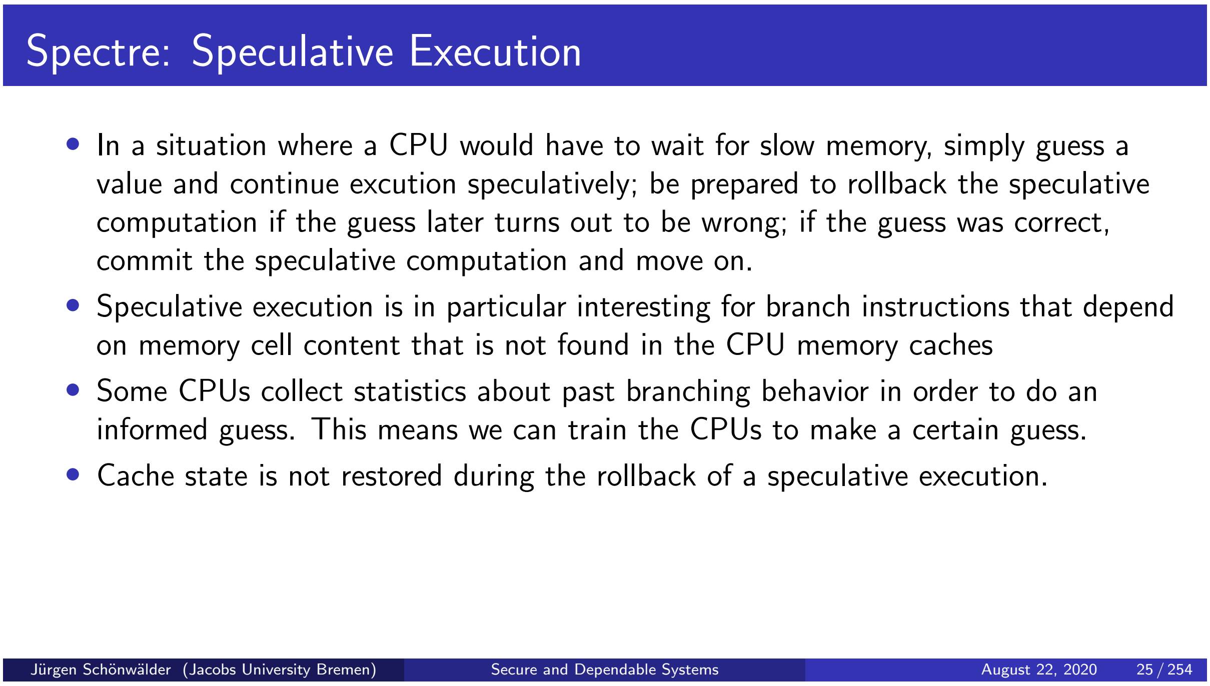


Spectre: Timing Side Channel Attack

- A side-channel attack is an attack where information is gained from the physical implementation of a computer system (e.g., timing, power consumption, radiation), 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:

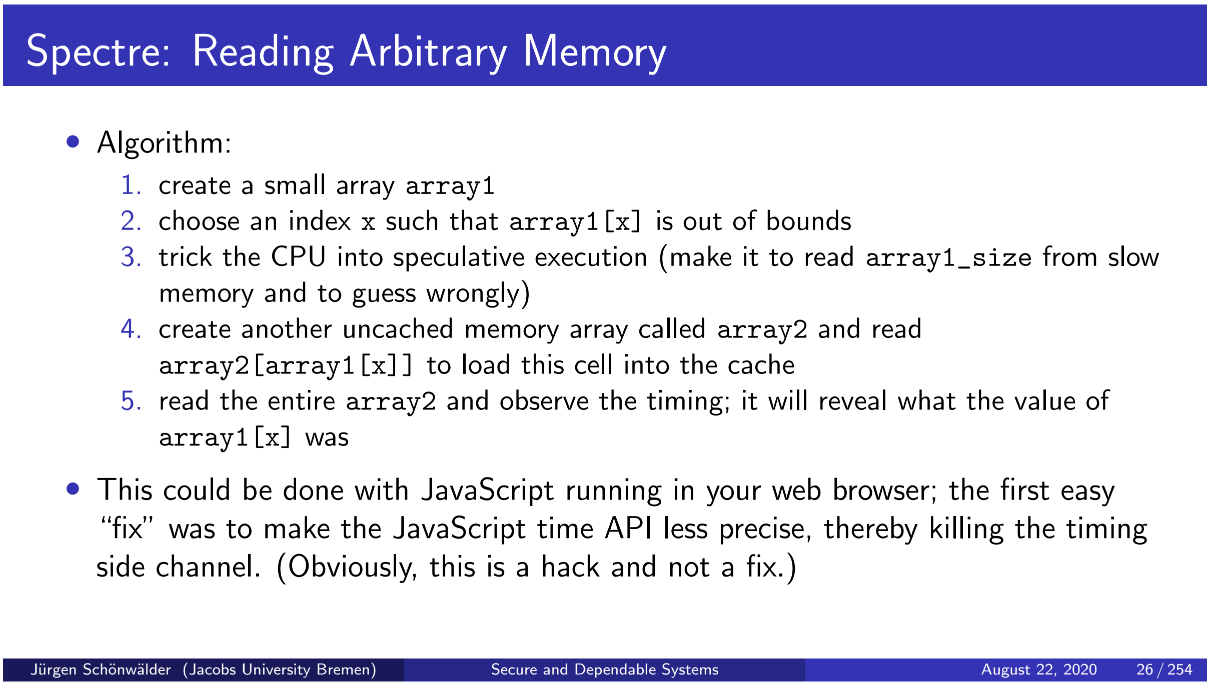
- The size of network packets can reveal which resources are accessed on a web server even if the communication is encrypted.
- The power consumption of displays can reveal what kind of content is displayed.
- Data transmitted over copper wires creates magnetic fields around the wire that can induce 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. (Obviously, this is a hack and not a fix.)

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, read [28] and [31] or take a look at the following videos:

YouTube: Spectre and Meltdown: Data leaks during speculative execution

YouTube: Spectre and Meltdown attacks explained understandably

Dependability Concepts and Terminology

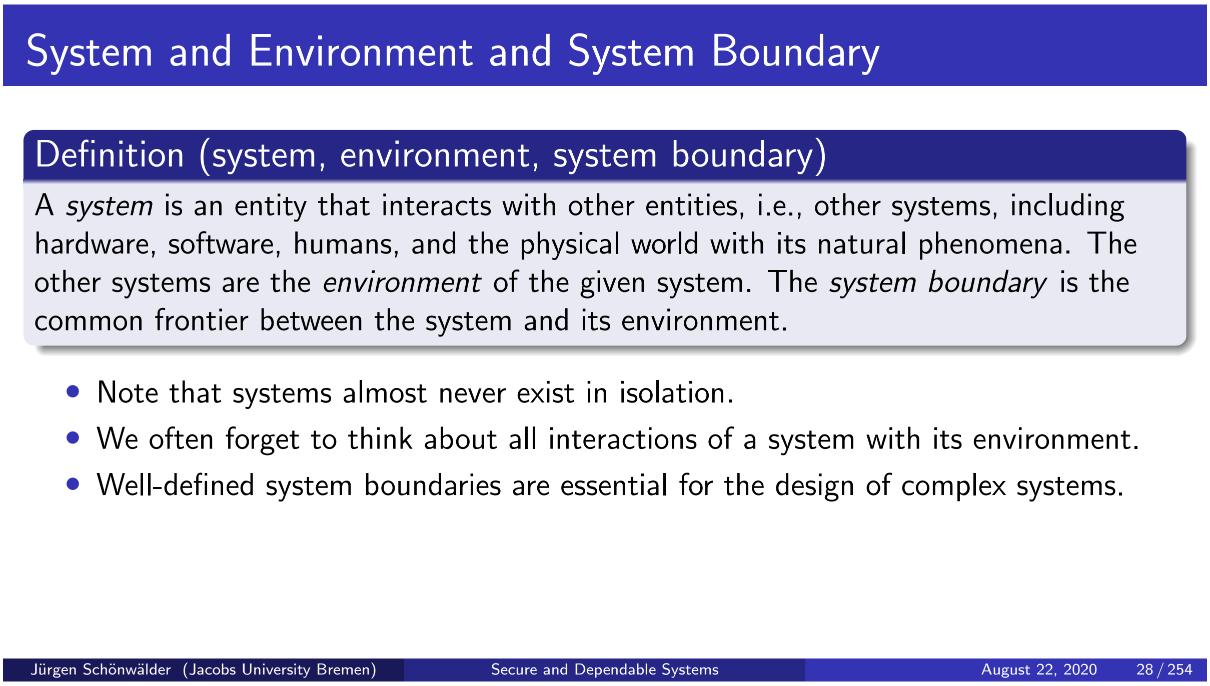
4 Motivation

5 Recent 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 [6]. Note that [6] 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)

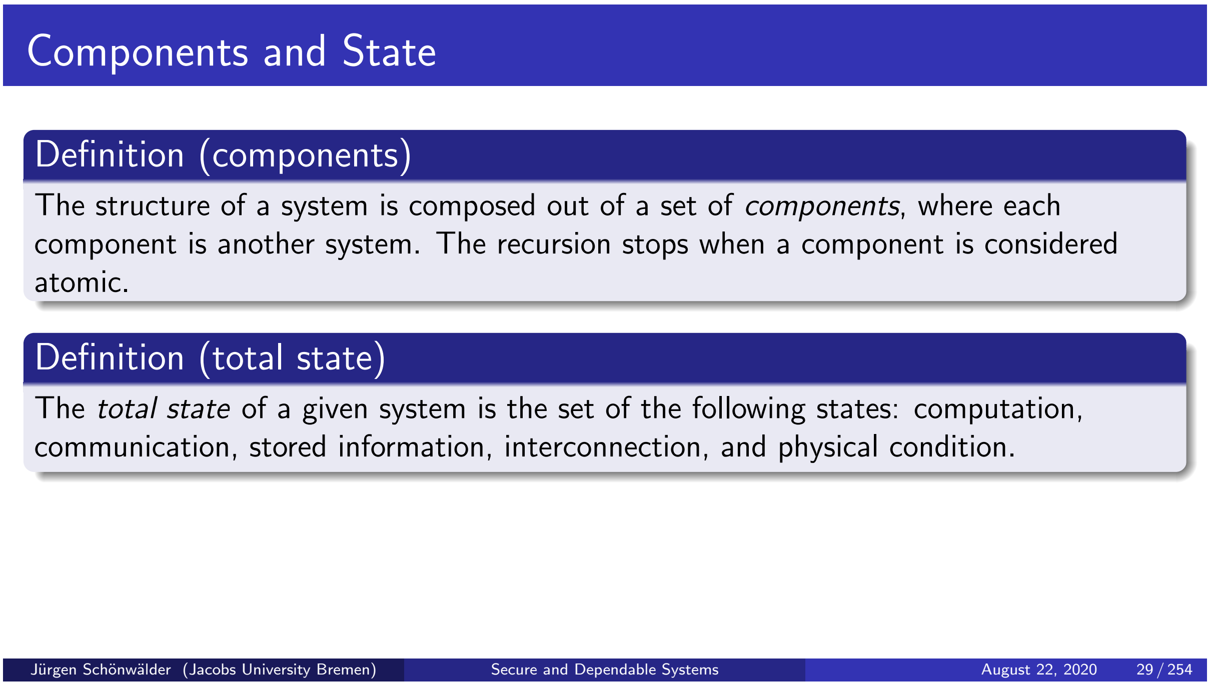
A *system* is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. The other systems are the *environment* of the given system. The *system boundary* is the common frontier between the system and its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



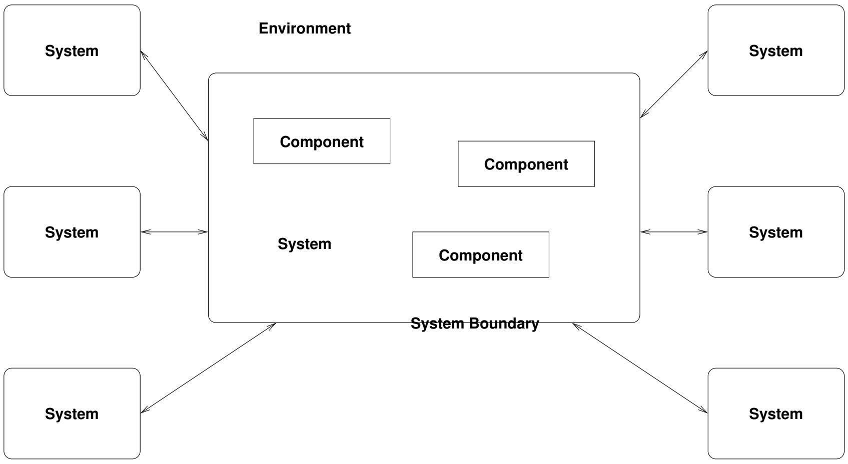
Components and State

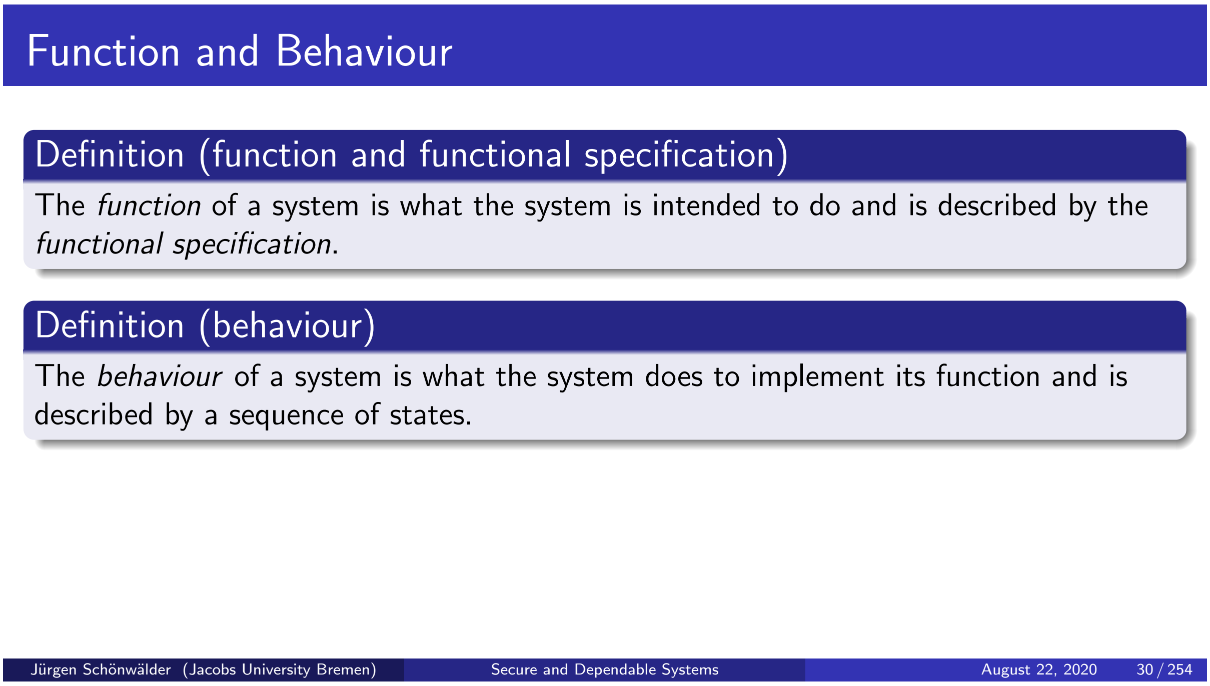
Definition (components)

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)

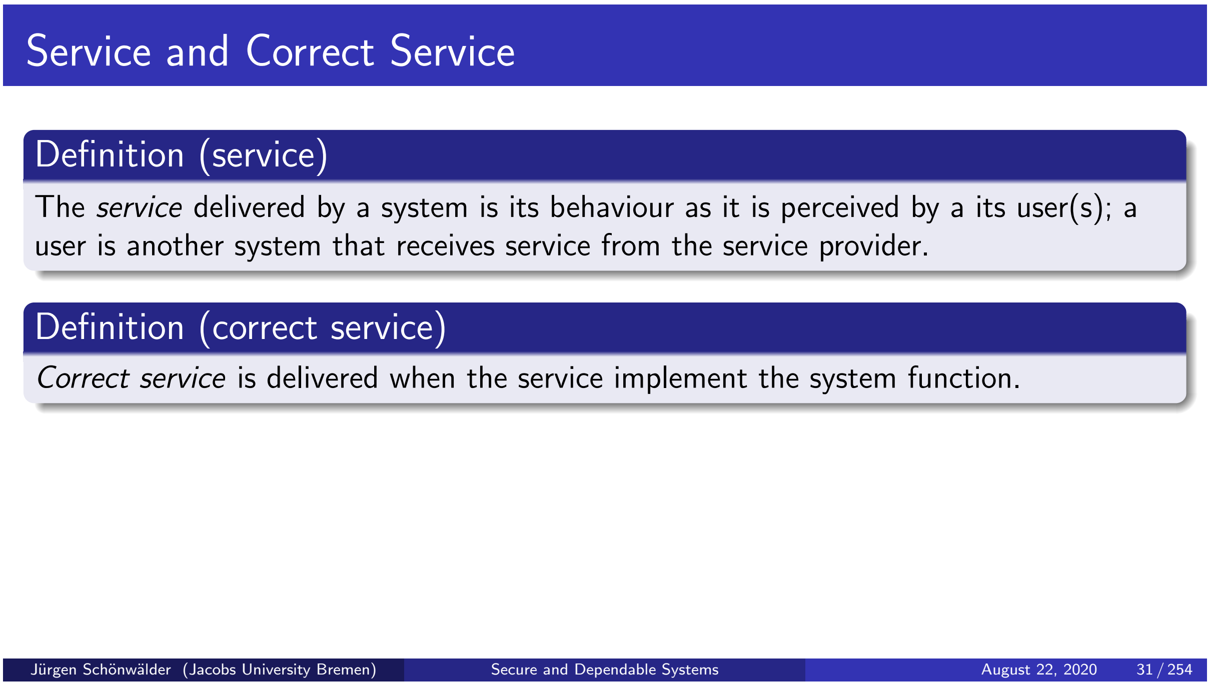
The *behaviour* of a system is what the system does to implement its function and is described by a sequence of states.

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

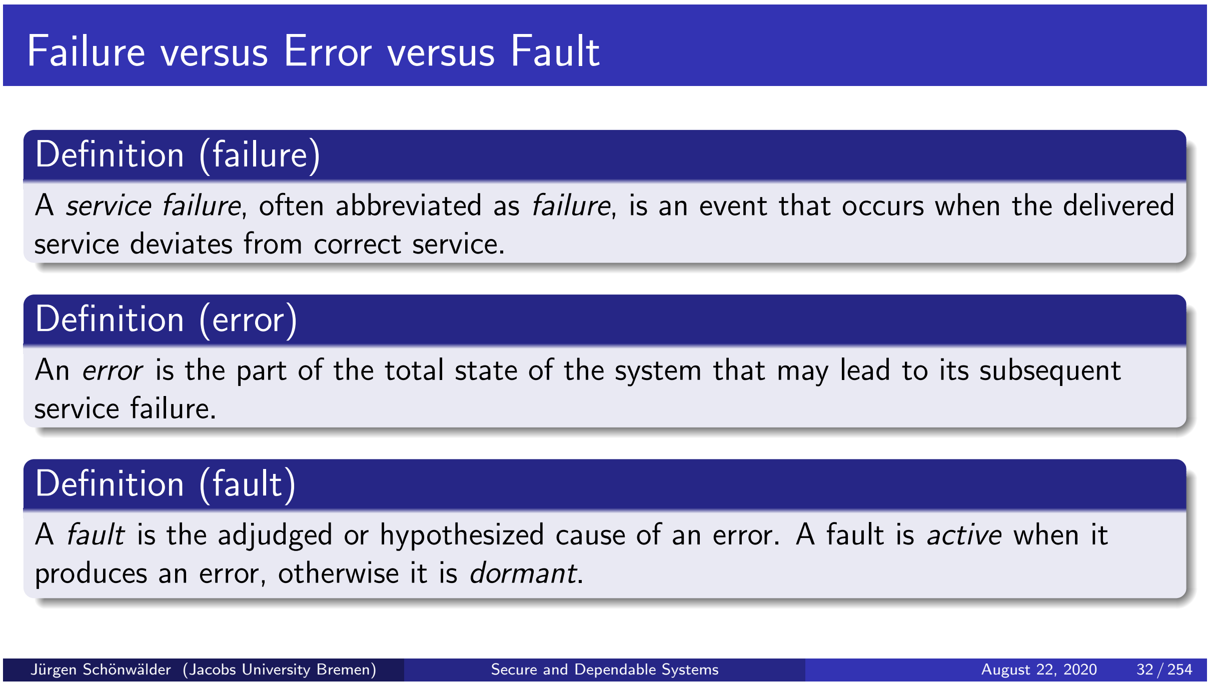
Definition (service)

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.

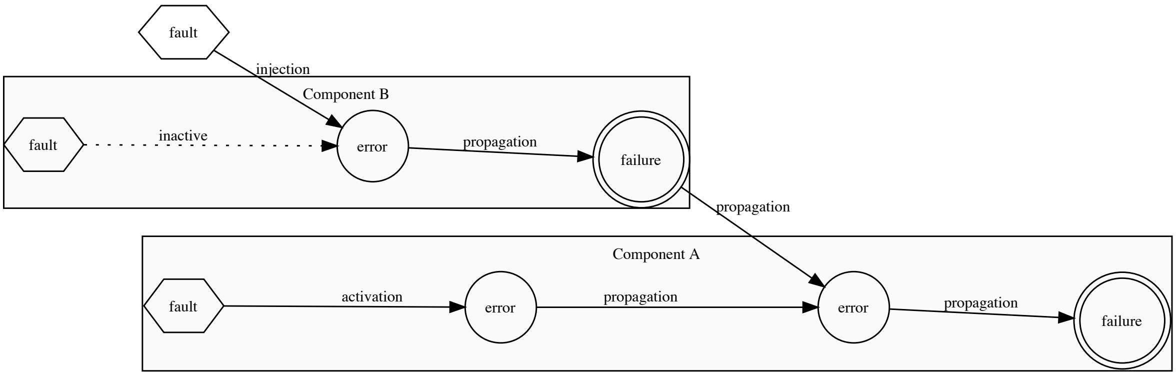
Definition (fault)

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

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 32 / 254

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 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

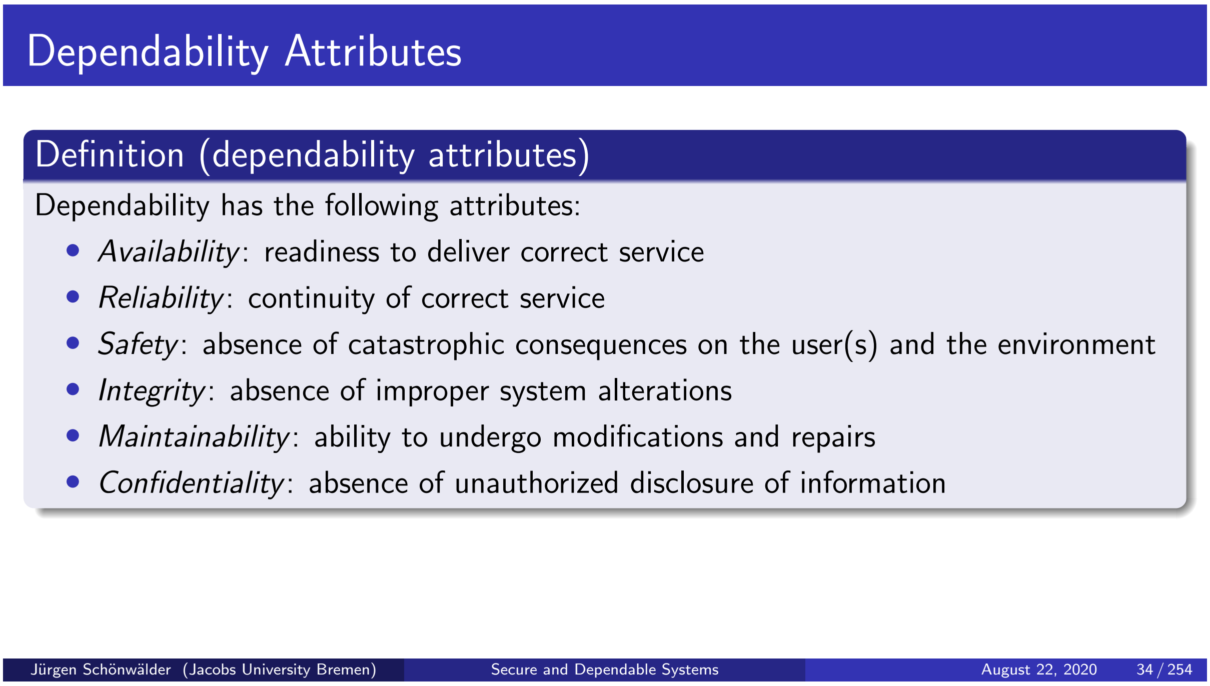
Definition (dependability - original)

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.

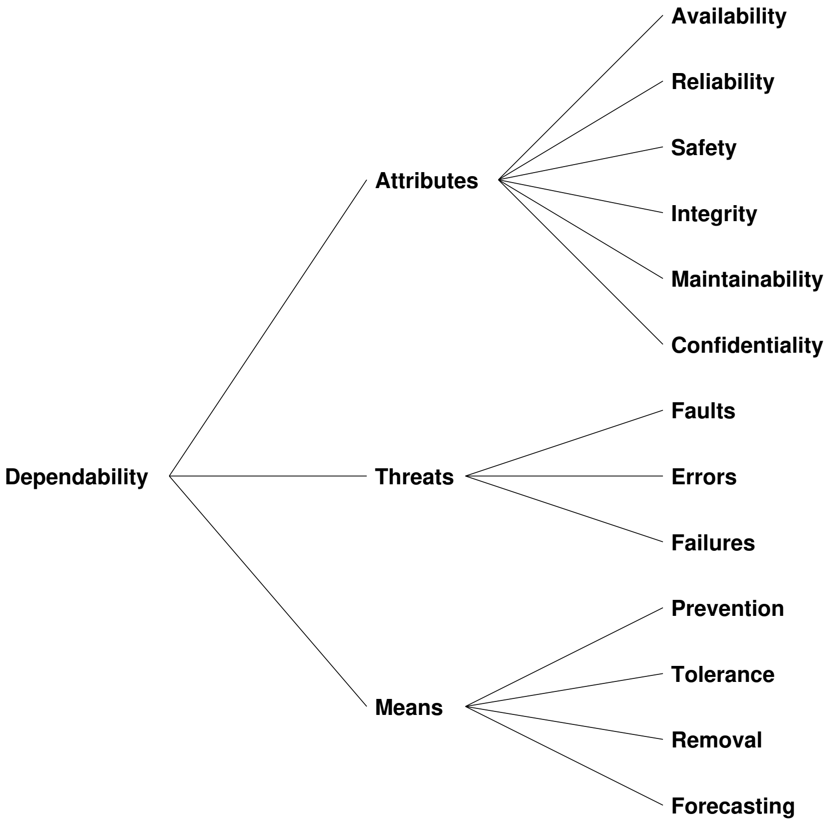


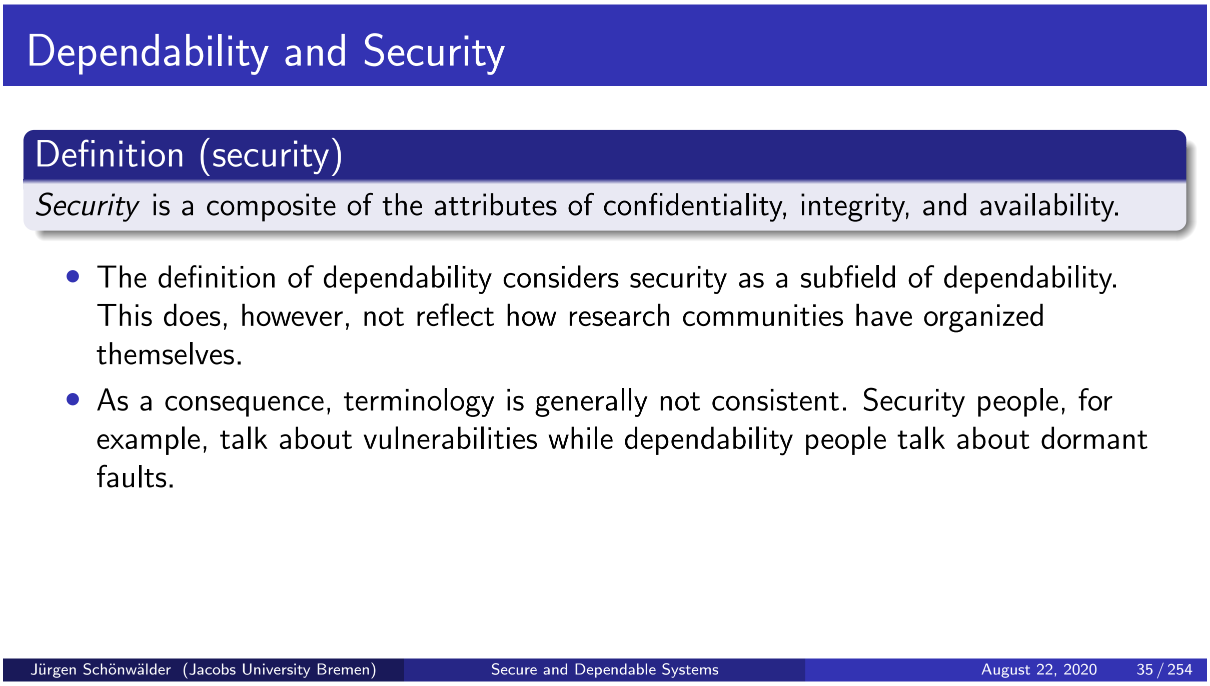
Dependability Attributes

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



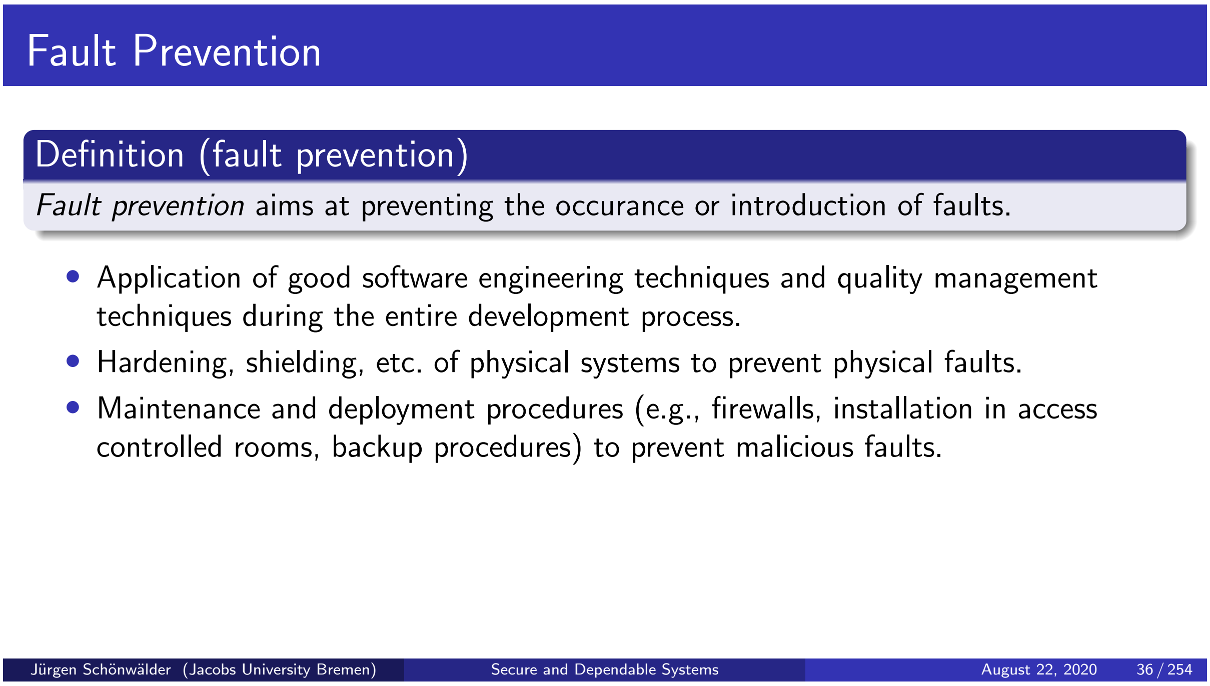


Dependability and Security

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)

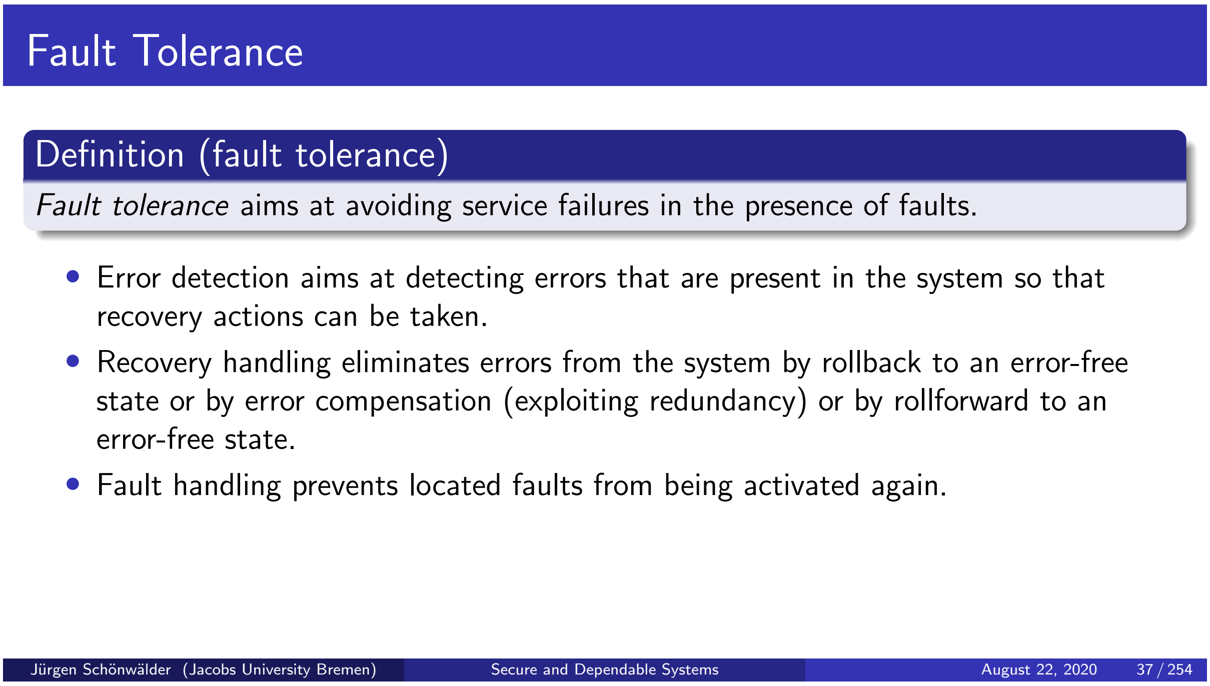
Fault prevention aims at preventing the occurance or introduction of faults.

- 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.

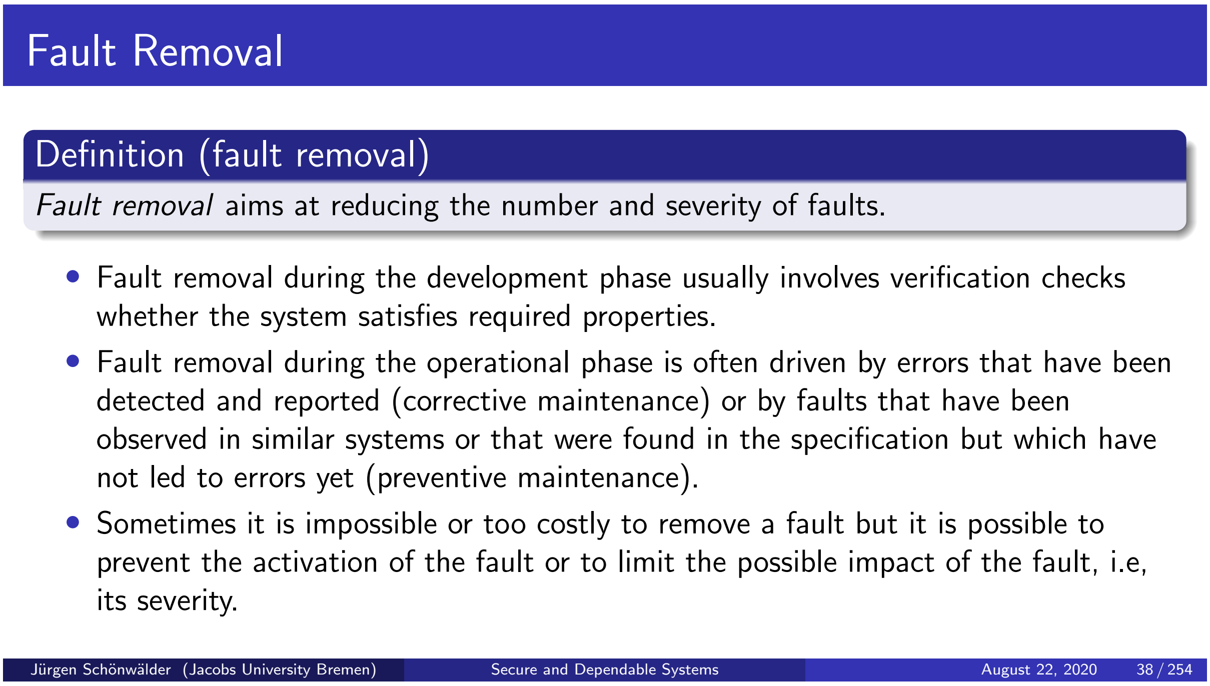
With the decreasing cost for computing power, it is meanwhile feasible to use replication of data and computations in order to produce redundancy that can be used to compensate errors and failures. For example, a query sent to a search engine may be given to multiple independent backend systems and the 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).

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.

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



Fault Removal

Definition (fault removal)

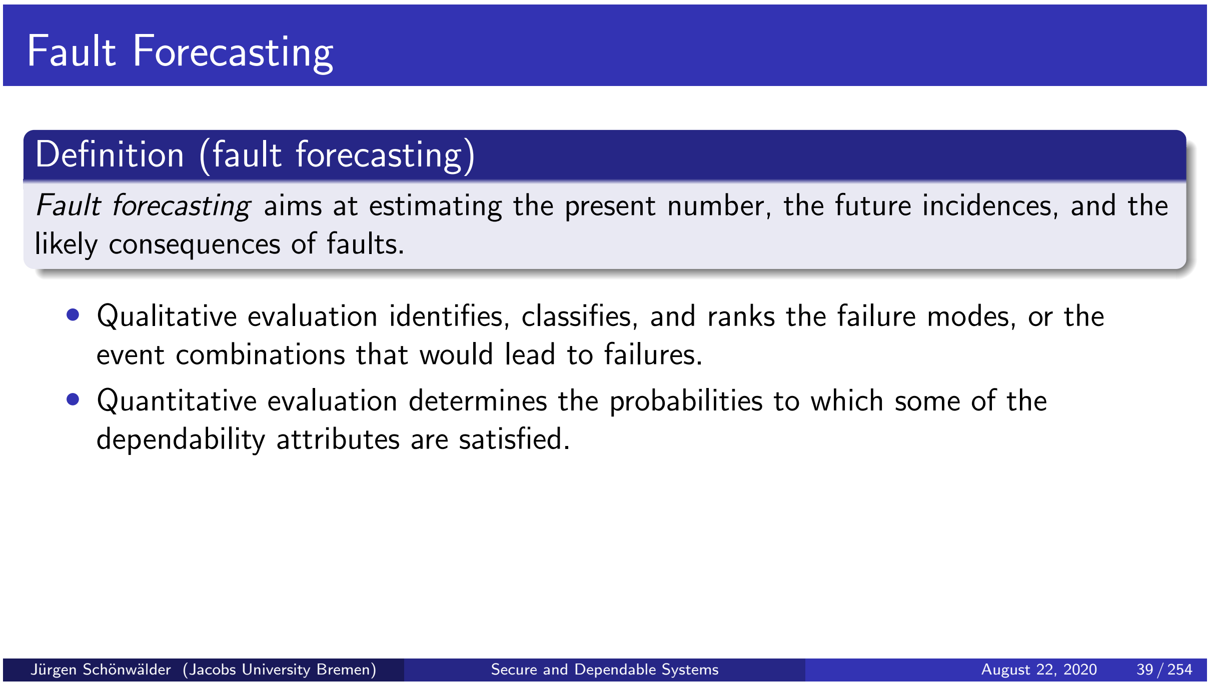
Fault removal aims at reducing the number and severity of faults.

- 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)

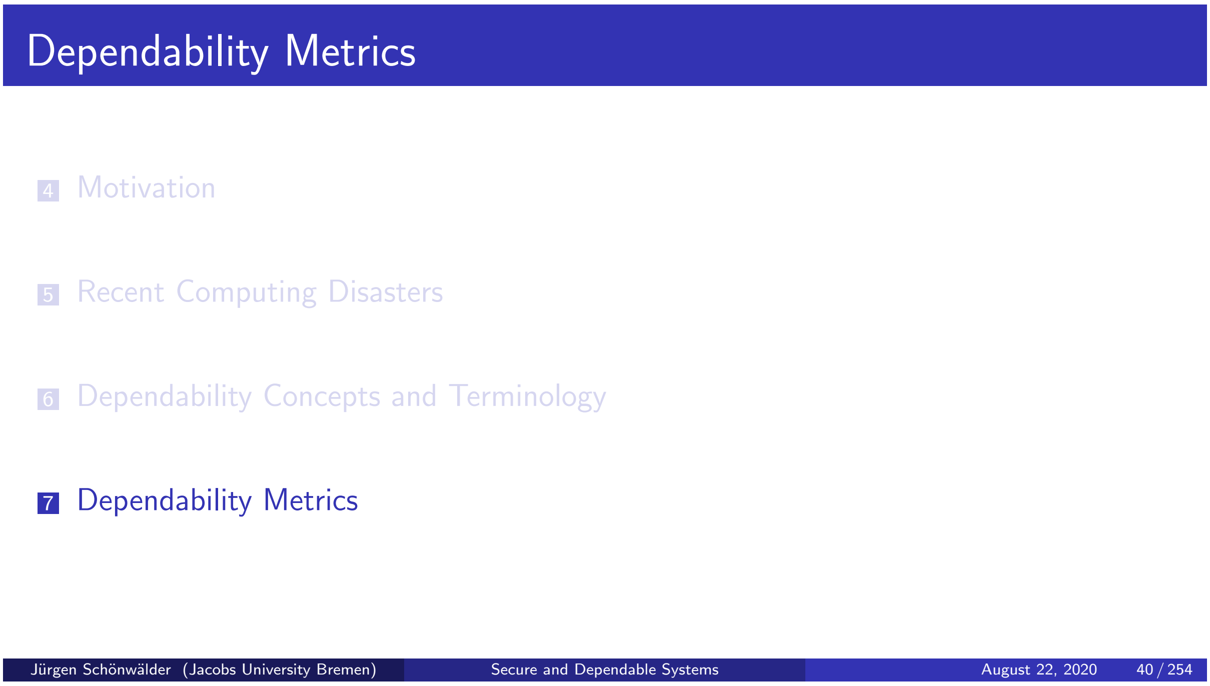
Fault forecasting aims at estimating the present number, the future incidences, and the likely consequences of faults.

- 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. The reason is that major new releases usually come with new bugs and it takes time to find and fix most of them.

Early adopters usually run higher risks to experience failures due to new faults added to a system.



Dependability Metrics

4 Motivation

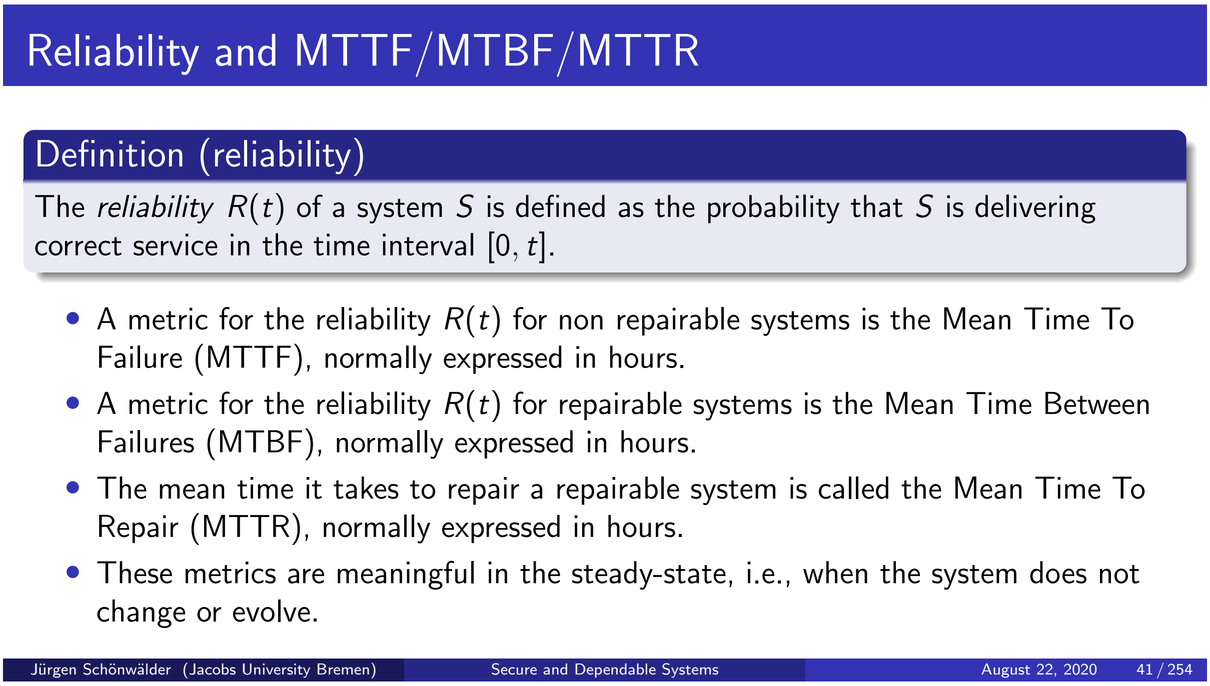
5 Recent Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 40 / 254

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

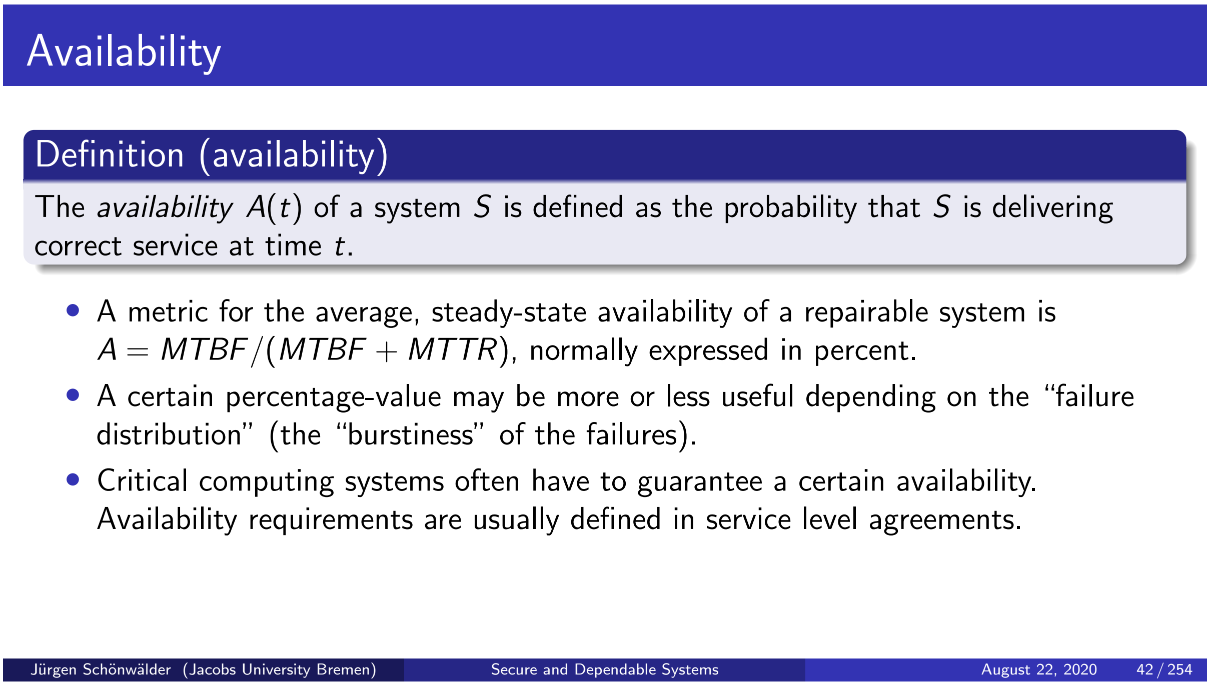


Reliability and MTTF/MTBF/MTTR

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 often 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.

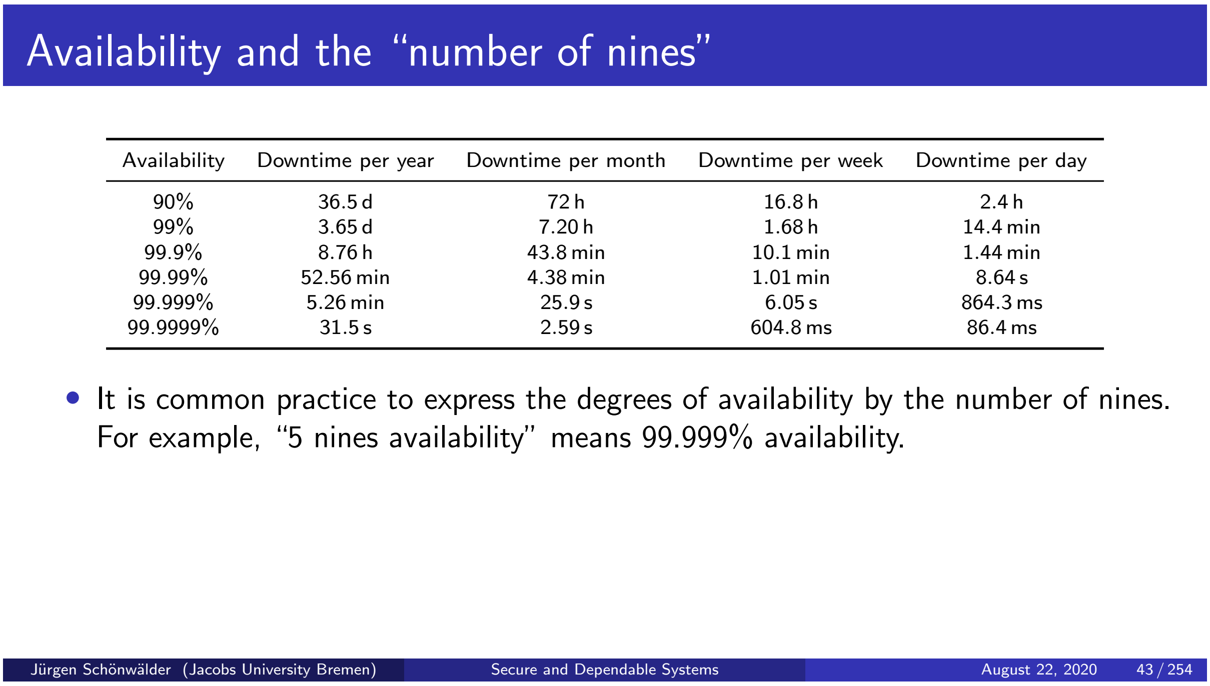
The Google Cloud Filestore Service Level Agreement for says (retrieved 2019-02-07):

> 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).



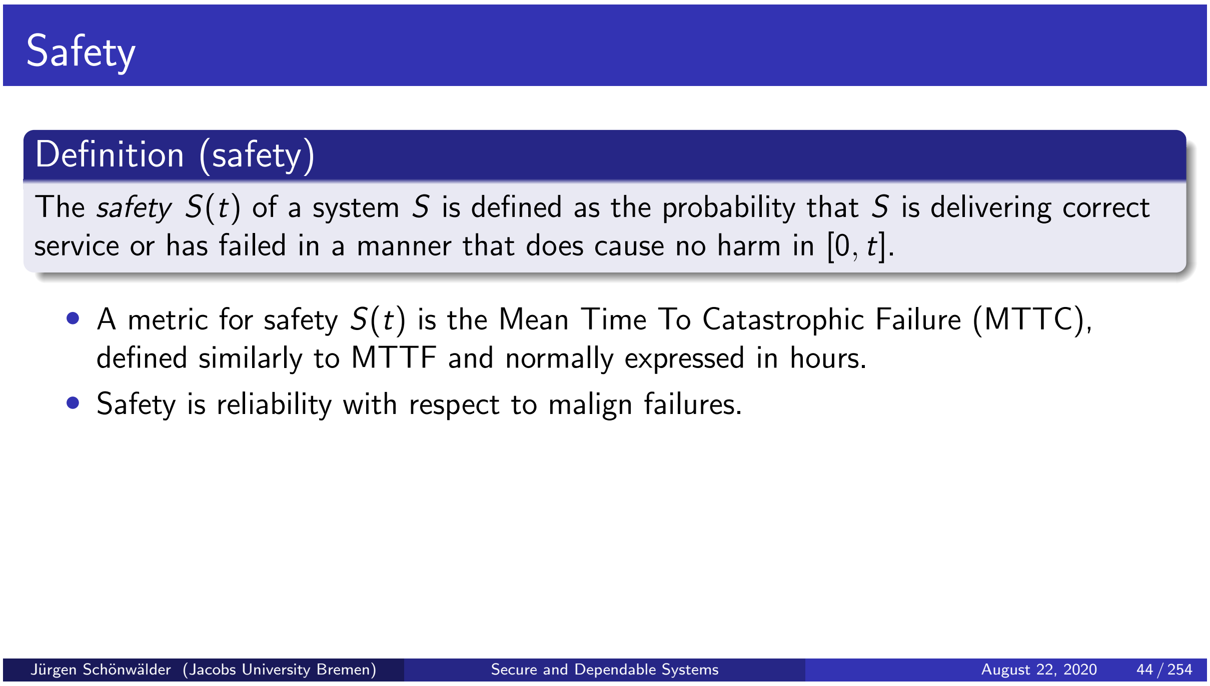
Availability and the "number of nines"

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 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 Aspects

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 excellent "Background reading on Hoare Logic".

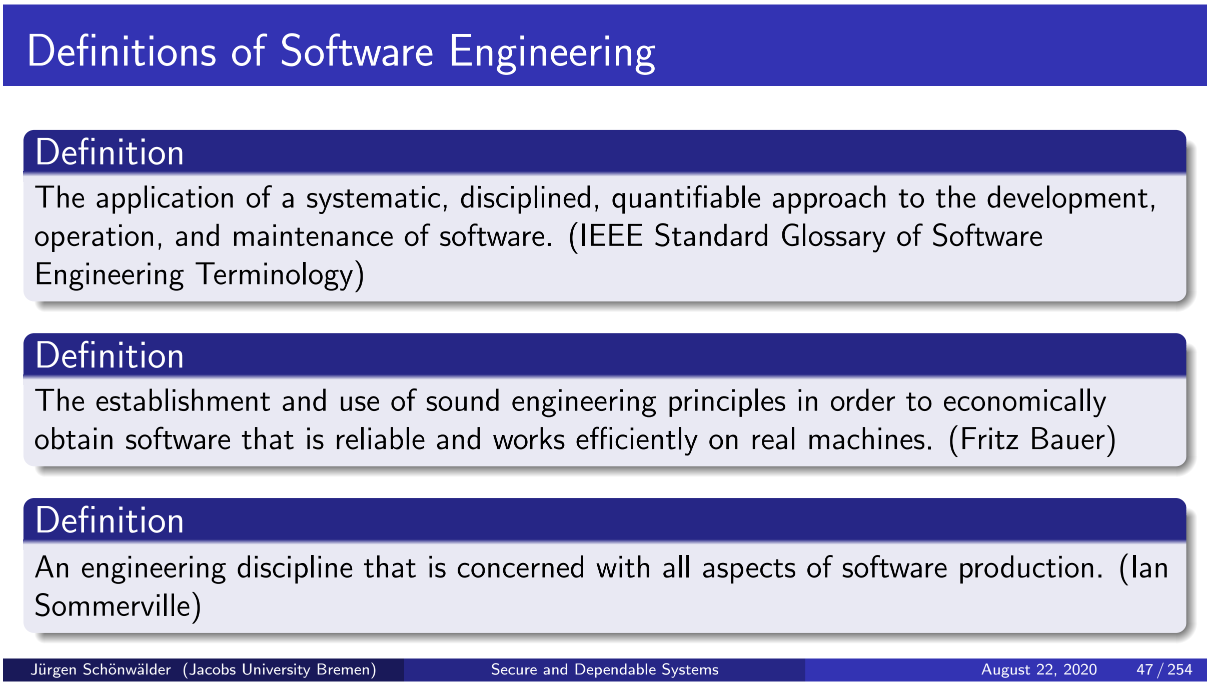
General Aspects

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)

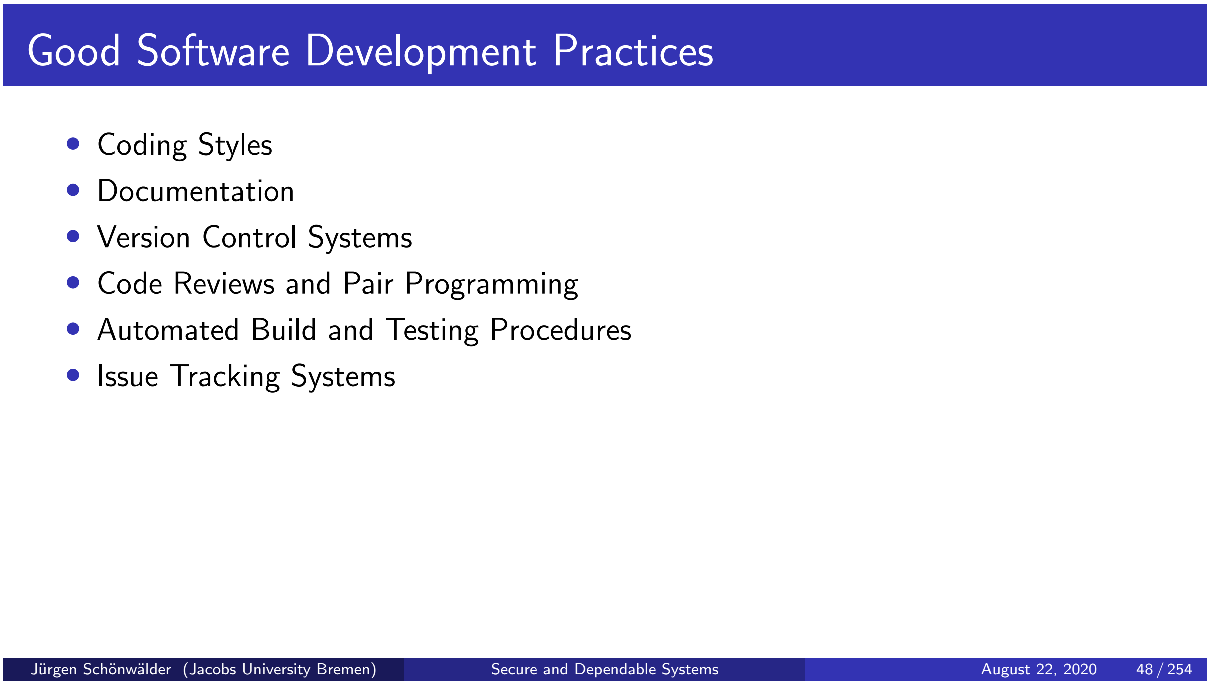
Definition

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.

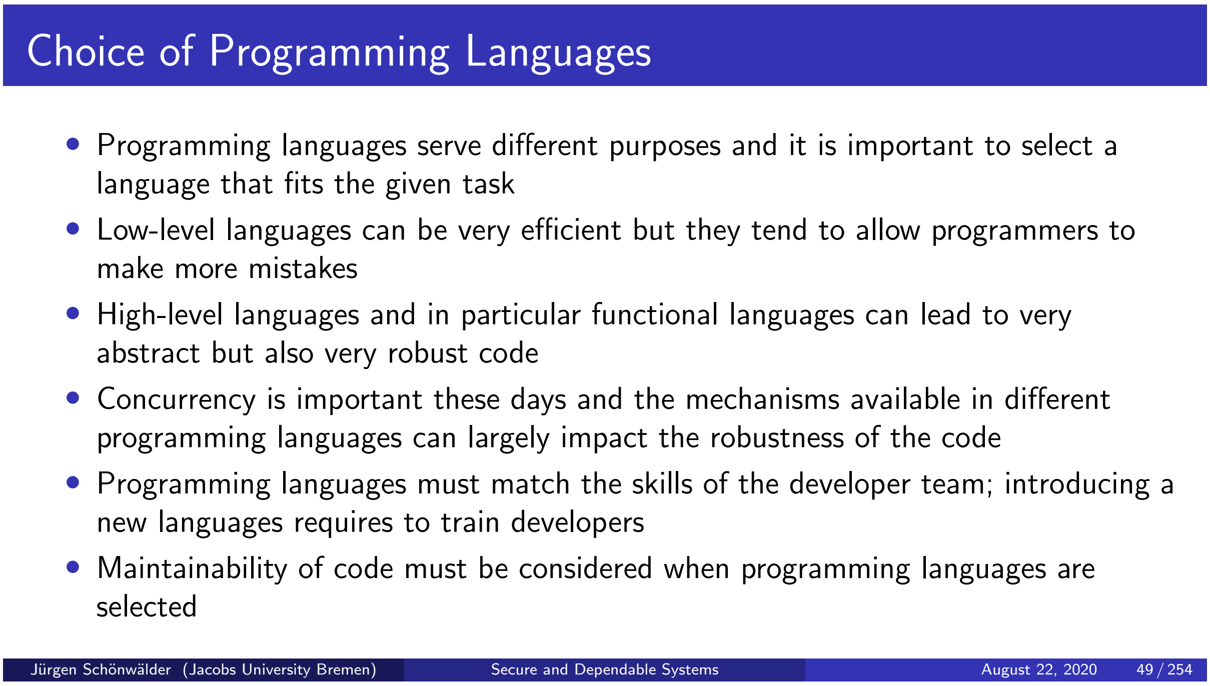
- Automated Build and Testing Procedures

  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.

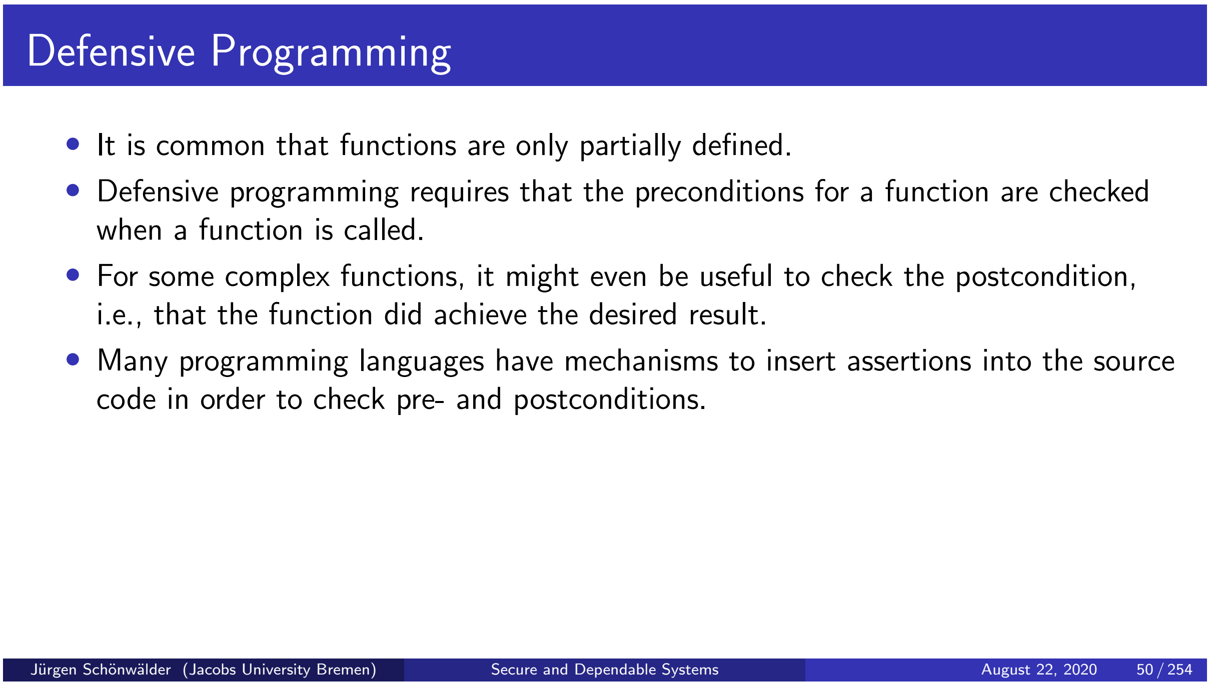
[1] https://en.wikipedia.org/wiki/MISRA_C
[2] https://en.wikipedia.org/wiki/Doxygen
[3] https://en.wikipedia.org/wiki/Git



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.

```c
#include <assert.h>

int average(int *a, int size)
{
    int sum = 0;
    assert(a && size > 0);
    for (int i = 0; i < size; i++) {
        sum += a[i];
    }
    return sum / size;
}
```

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:

```c
#include <assert.h>

void sort(int *a, size_t n)
{
    assert(a);
    recursive_super_duper_sort(a, 0, n);
    assert(is_sorted(a, n));
}

static bool is_sorted(int const *a, size_t n)
{
    for (size_t i=0; i<n-1; i++) {
        if (a[i] > a[i+1]) { return false; }
    }
    return true;
}

int binary_search(int *a, size_t n, int value)
{
    assert(a && is_sorted(a, n));
    // ...
}
```

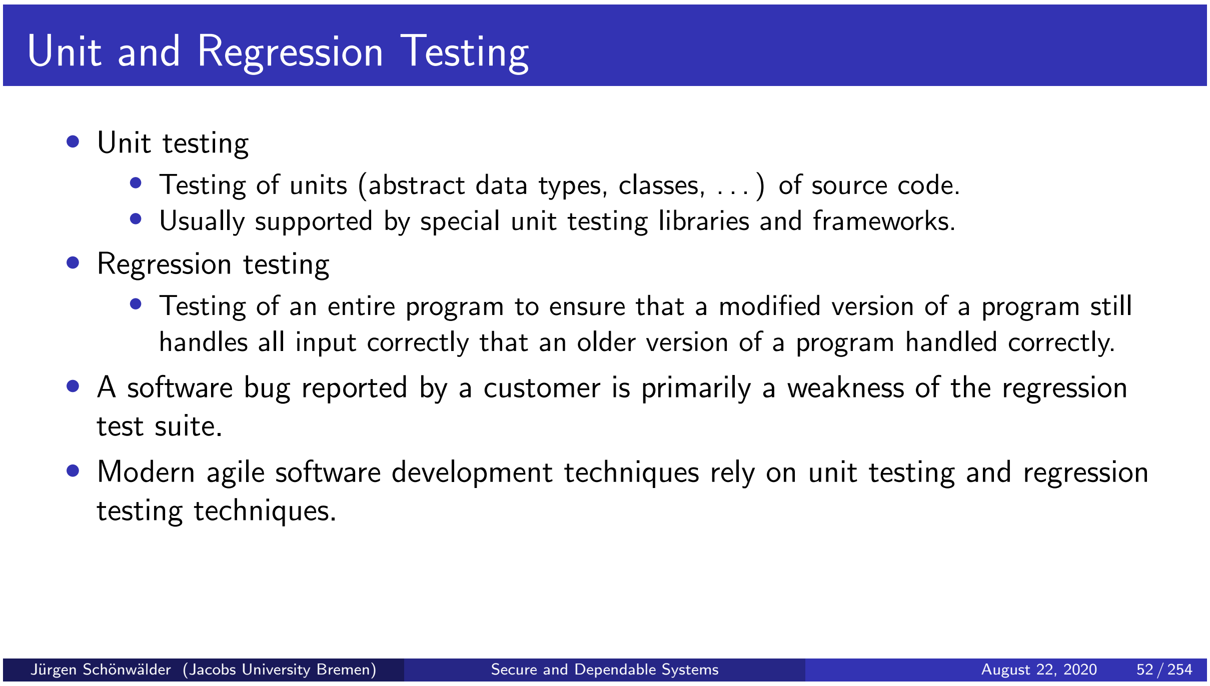
Software Testing

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.

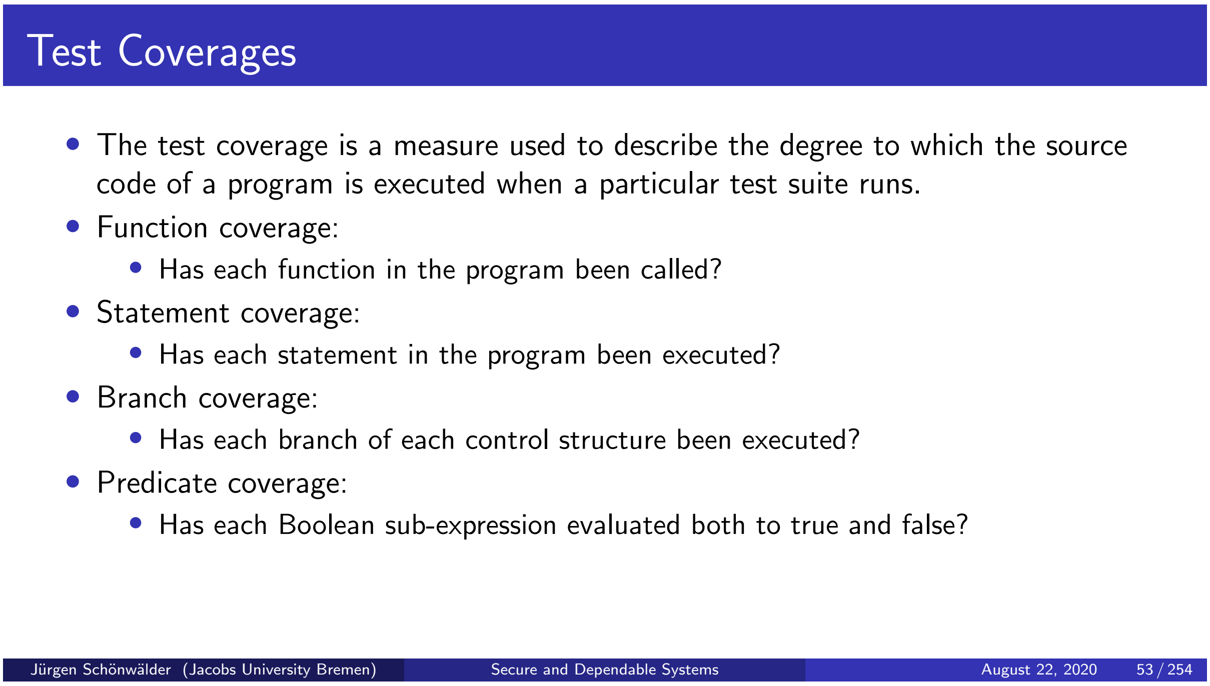
Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 52 / 254

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:

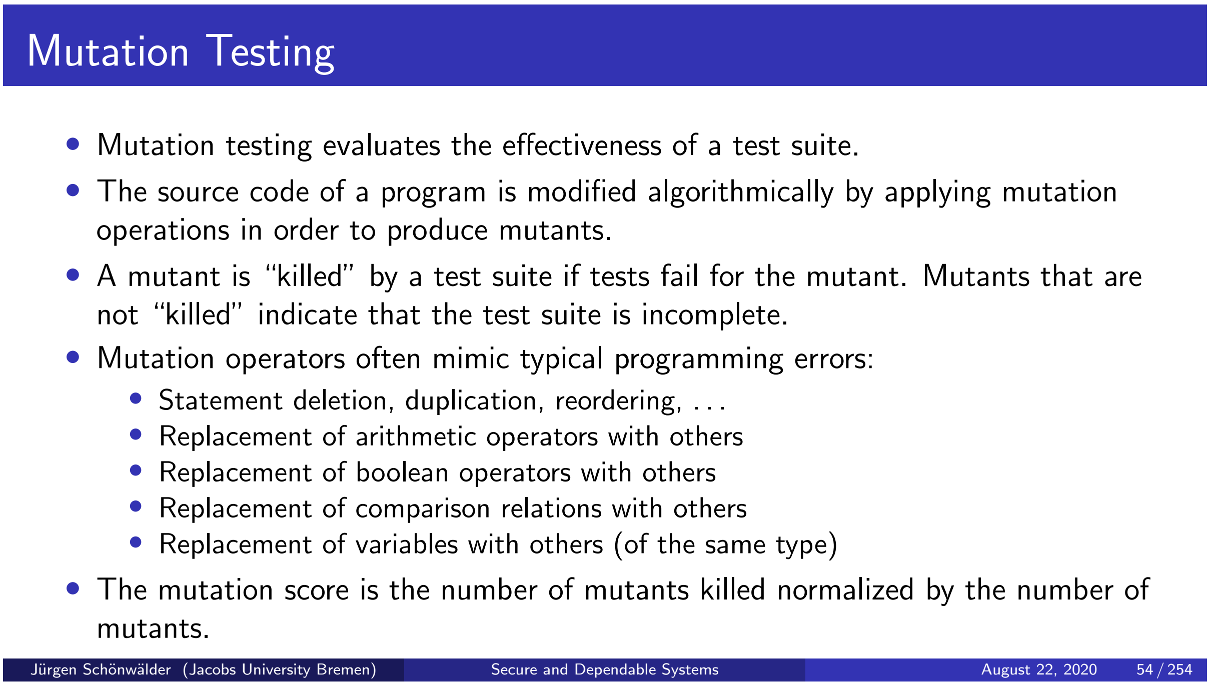
- check unit testing framework for C
- cmocka unit testing framework for C
- Catch^2 automated test framework for C++
- Google Test Google's C++ test framework
- junit Java unit testing framework that inspired many other testing frameworks



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:
  - 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?

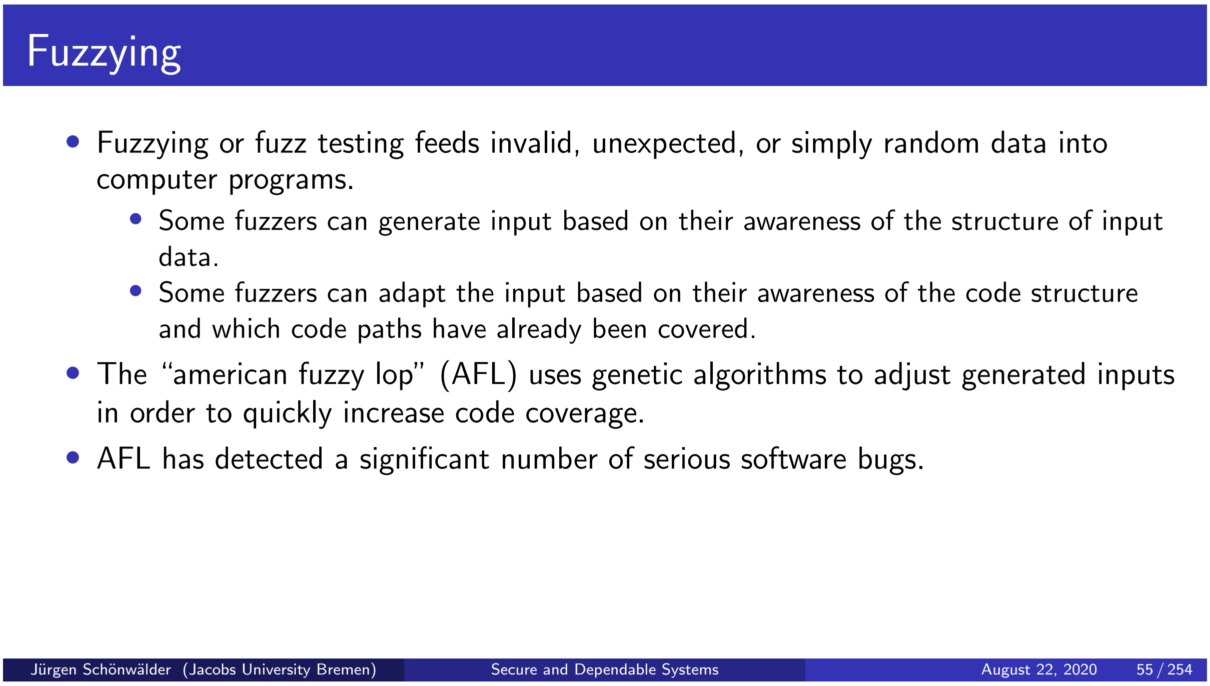
A common approach to obtain test coverage information is to instrument a program at compilation time (using special compiler options) so that coverage information is collected when the program is executed. The collected raw information can then subsequently be analyzed to obtain coverage reports.



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 operators with others
  - Replacement of boolean operators 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.



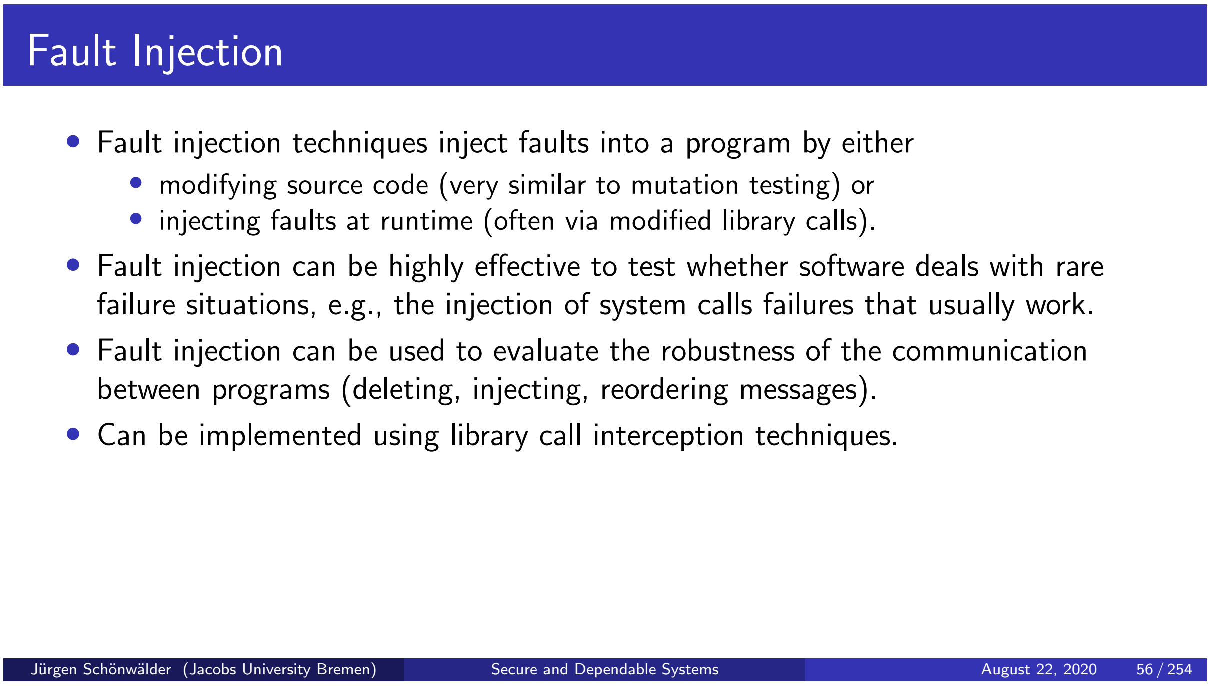
Fuzzying

- 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

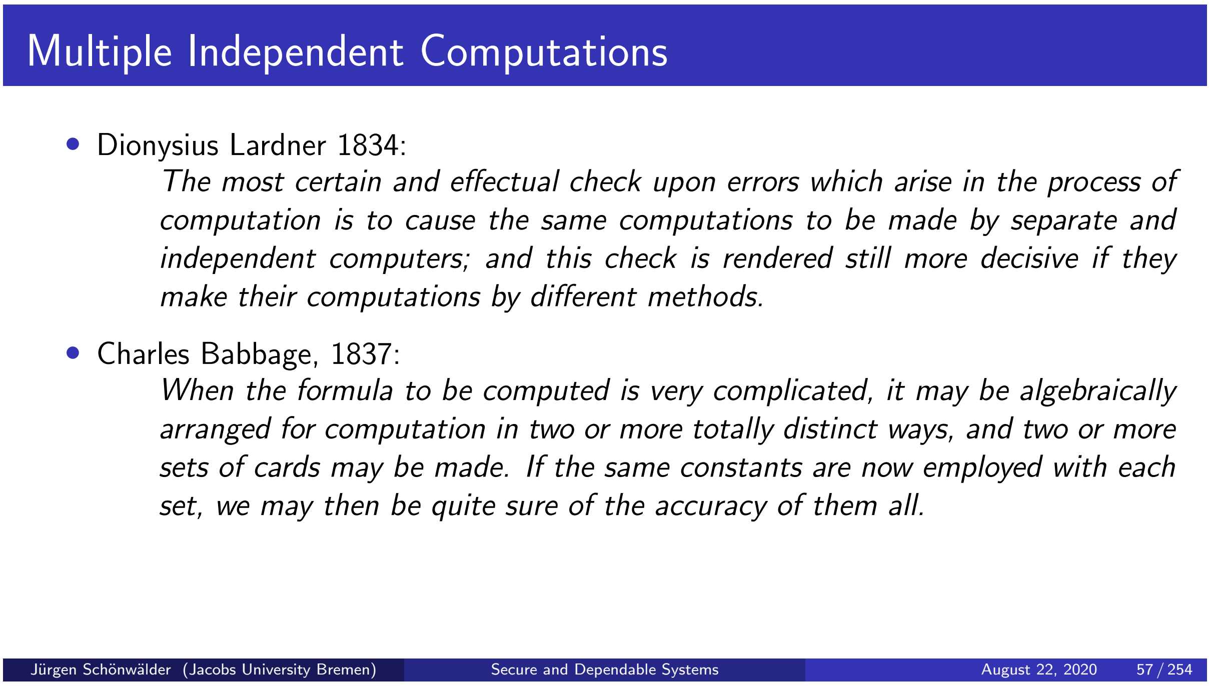
- Fault injection techniques inject faults into a program by either
  - modifying source code (very similar to mutation testing) or
  - injecting faults at runtime (often via modified library calls).
- Fault injection can be highly effective to test whether software deals with rare failure situations, 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 [22]. 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).

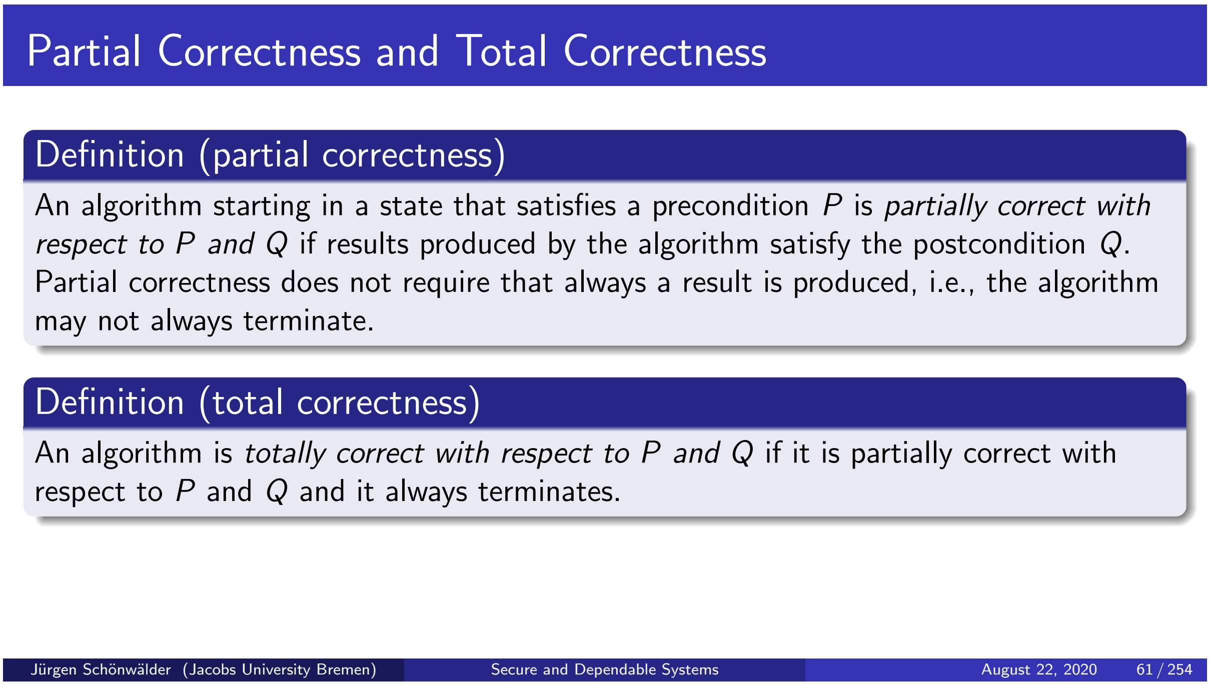
P and Q are conditions on program variables. They will be written using standard mathematical notation and logical operators. The predicate P defines the subset of all possible states for which a program C is defined. Similarly, the predicate Q defines the subset of all possible states for which the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

It is possible that different programs satisfy the same specification:

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

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

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



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.

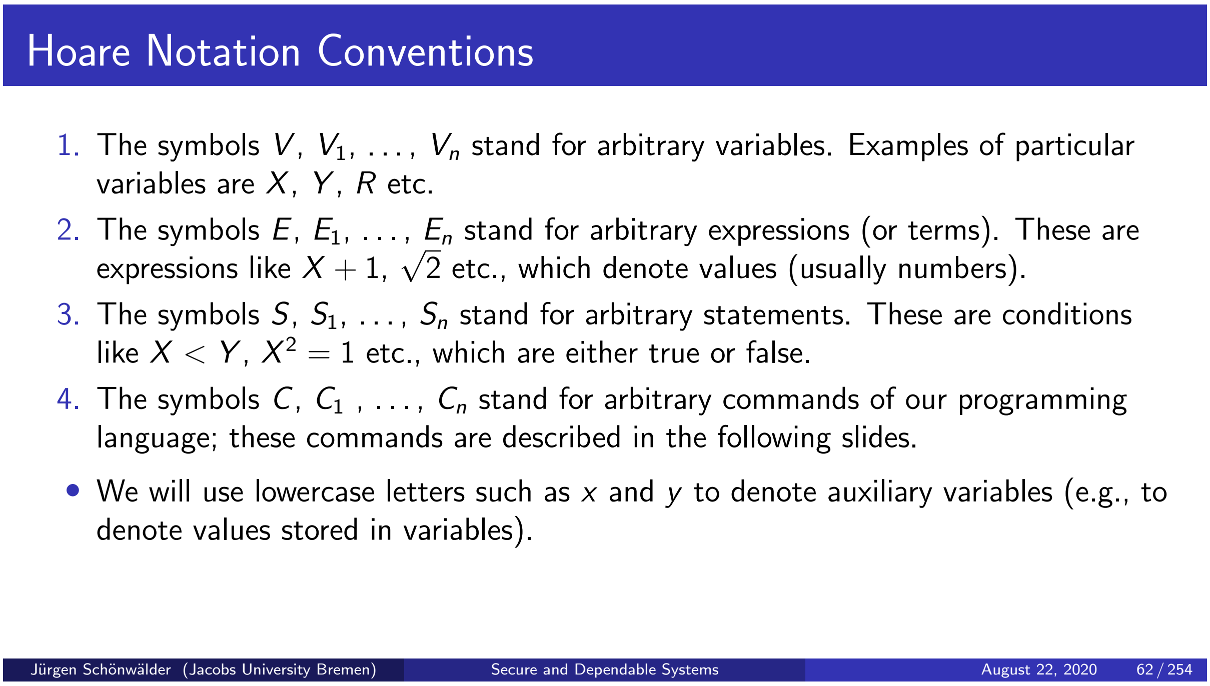
Definition (total correctness)

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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 61 / 254

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

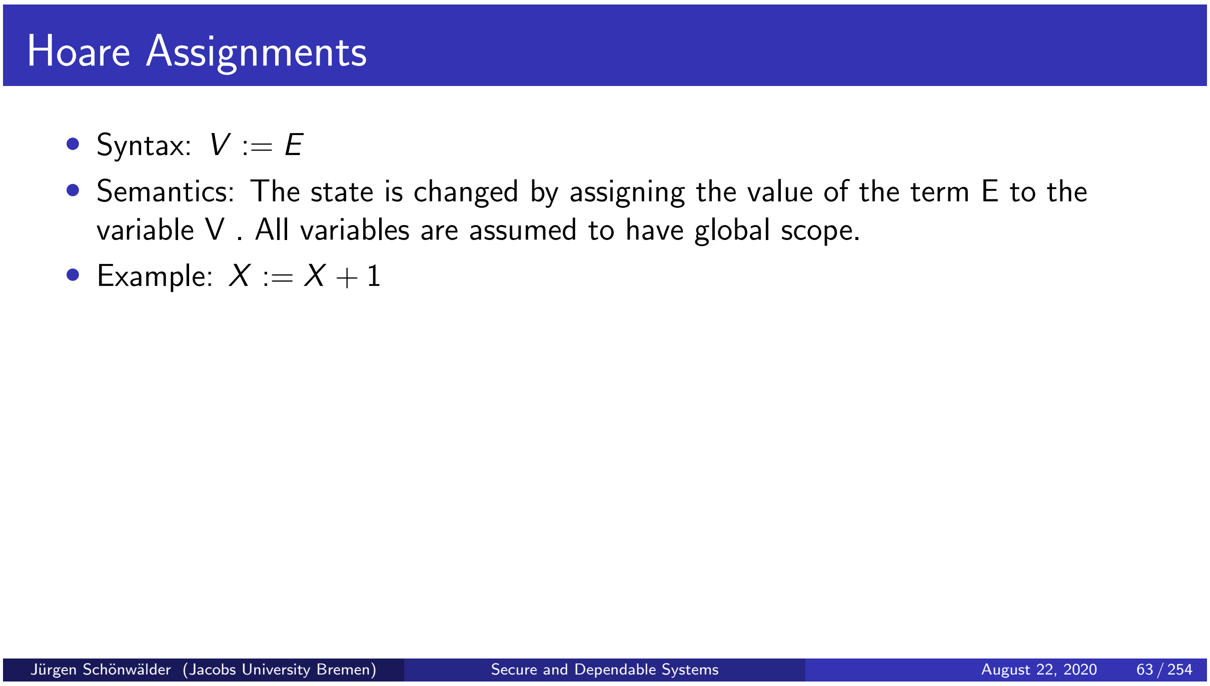
1. The symbols V, V_1, $\ldots$, V_n stand for arbitrary variables. Examples of particular variables are X, Y, R etc.
2. The symbols E, E_1, $\ldots$, E_n stand for arbitrary expressions (or terms). These are expressions like $X + 1$, $\sqrt{2}$ etc., which denote values (usually numbers).
3. The symbols S, S_1, $\ldots$, 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 , $\ldots$, 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).

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 62 / 254

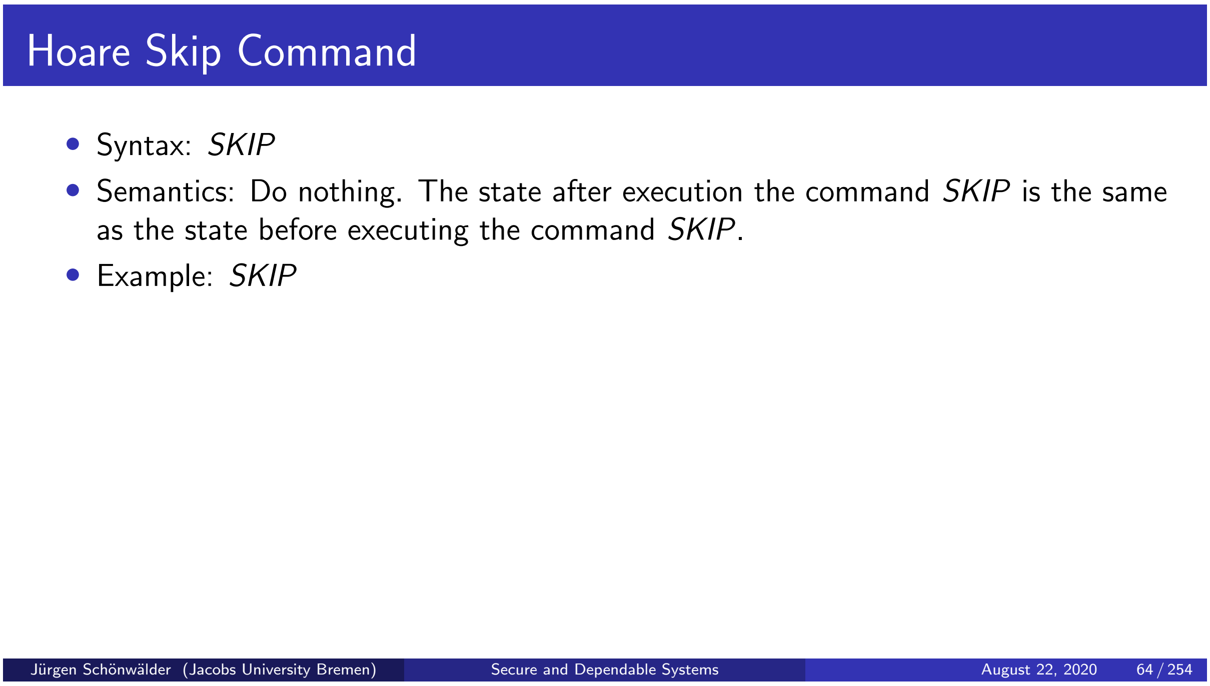
We are focusing in the following on a purely imperative programming model where a global set of variables determines the current state of the computation. A subset of the variables are used to provide the input to an algorithm and another subset of the variables provides the output of an algorithm.

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

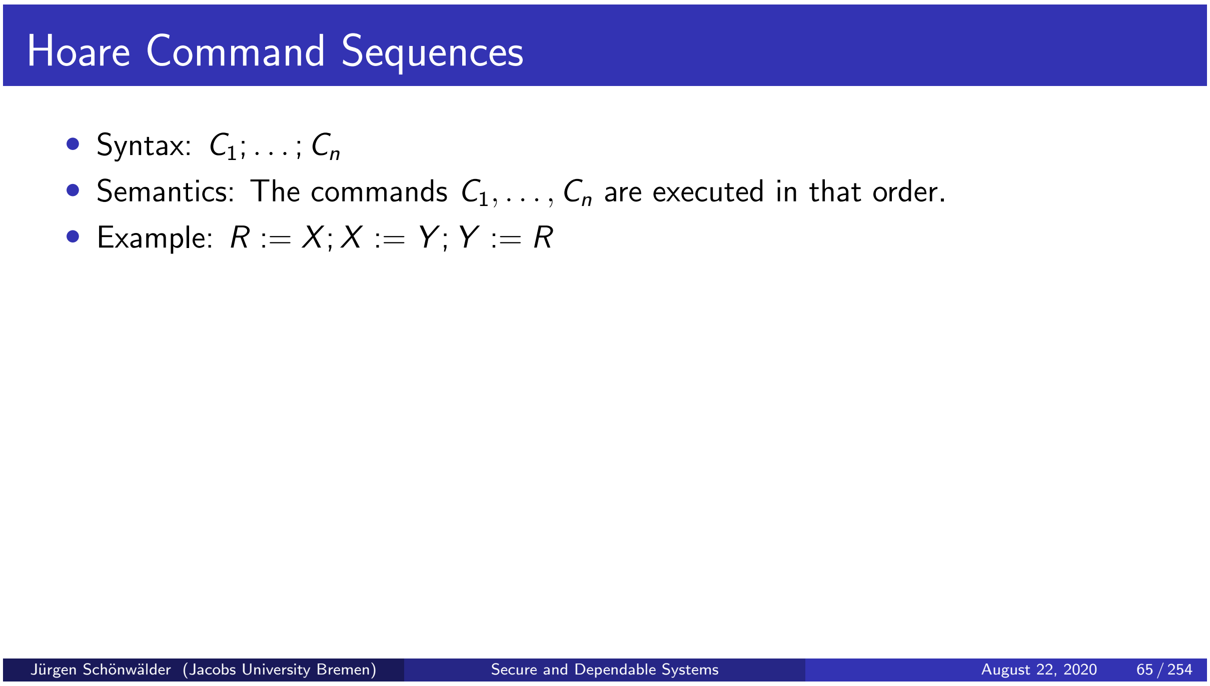
- Syntax: $V := E$
- Semantics: The state is changed by assigning the value of the term E to the variable V . All variables are assumed to have global scope.
- Example: $X := X + 1$



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

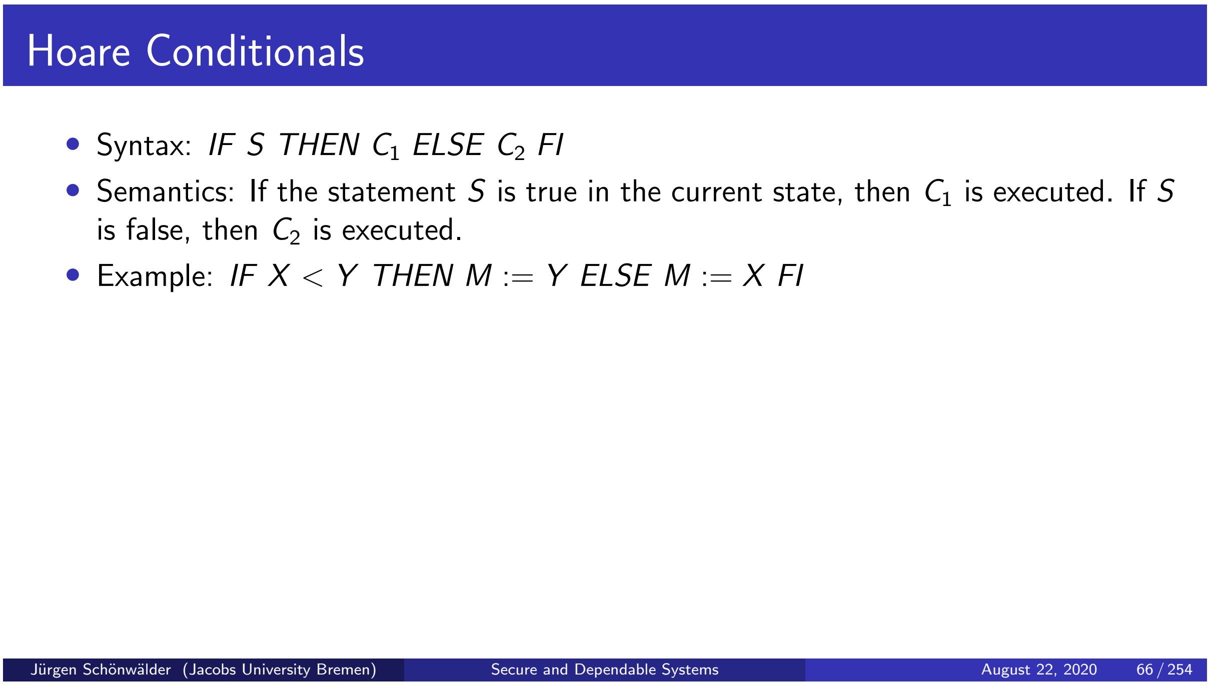
- Syntax: $C_1; \dots; C_n$
- Semantics: The commands $C_1, \dots, C_n$ are executed in that order.
- Example: $R := X; X := Y; Y := R$

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:

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

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

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



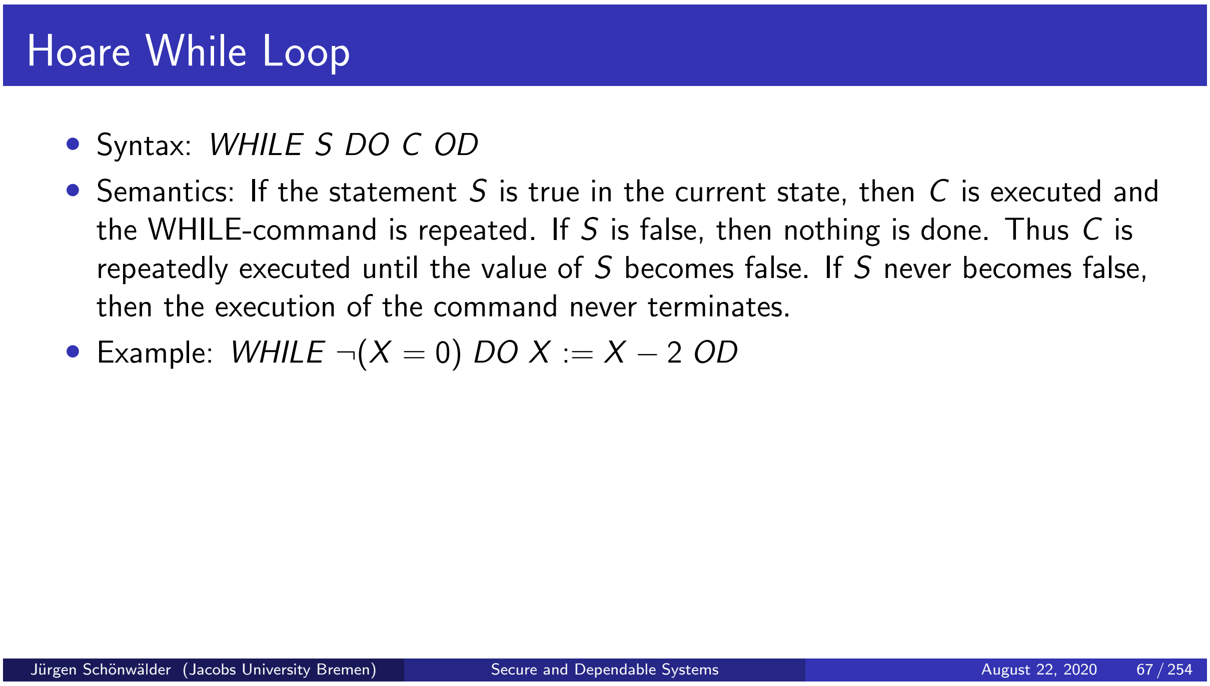
Hoare Conditionals

- 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

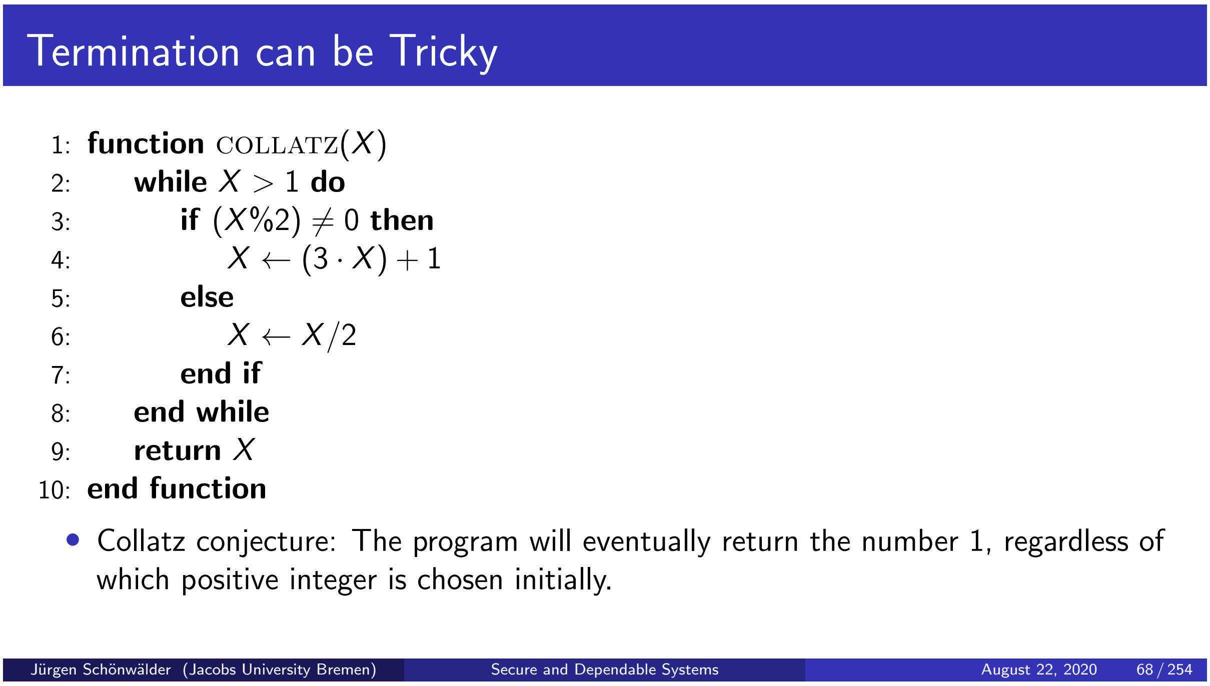
IF S THEN SKIP ELSE C FI



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: $\mathit{WHILE}\ \neg(X = 0)\ \mathit{DO}\ X := X - 2\ \mathit{OD}$

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

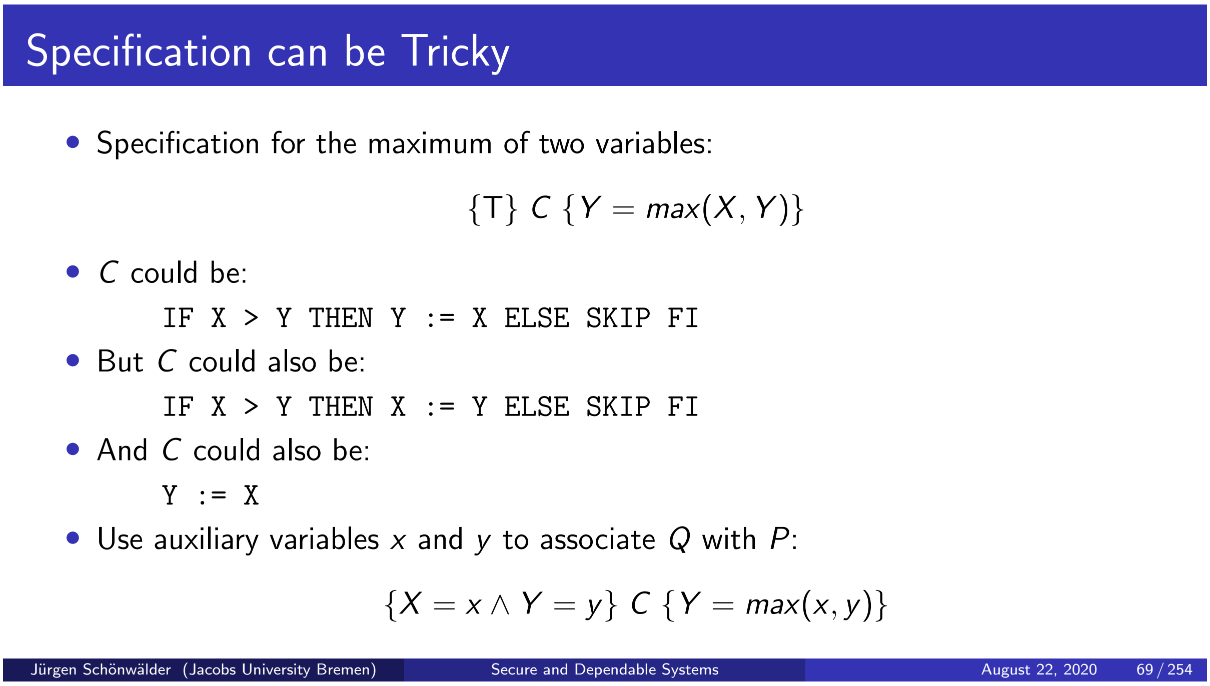
```
function COLLATZ($X$)
    while $X > 1$ do
        if $(X\%2) \neq 0$ then
            $X \leftarrow (3 \cdot X) + 1$
        else
            $X \leftarrow X/2$
        end if
    end while
    return $X$
end function
```

- 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:

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

- 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: $\{X > 0 \wedge X = x\}$
$F := 1$
while $X > 0$ do
    $F := F \cdot X$
    $X := X - 1$
od
Postcondition: $\{F = x!\}$
```

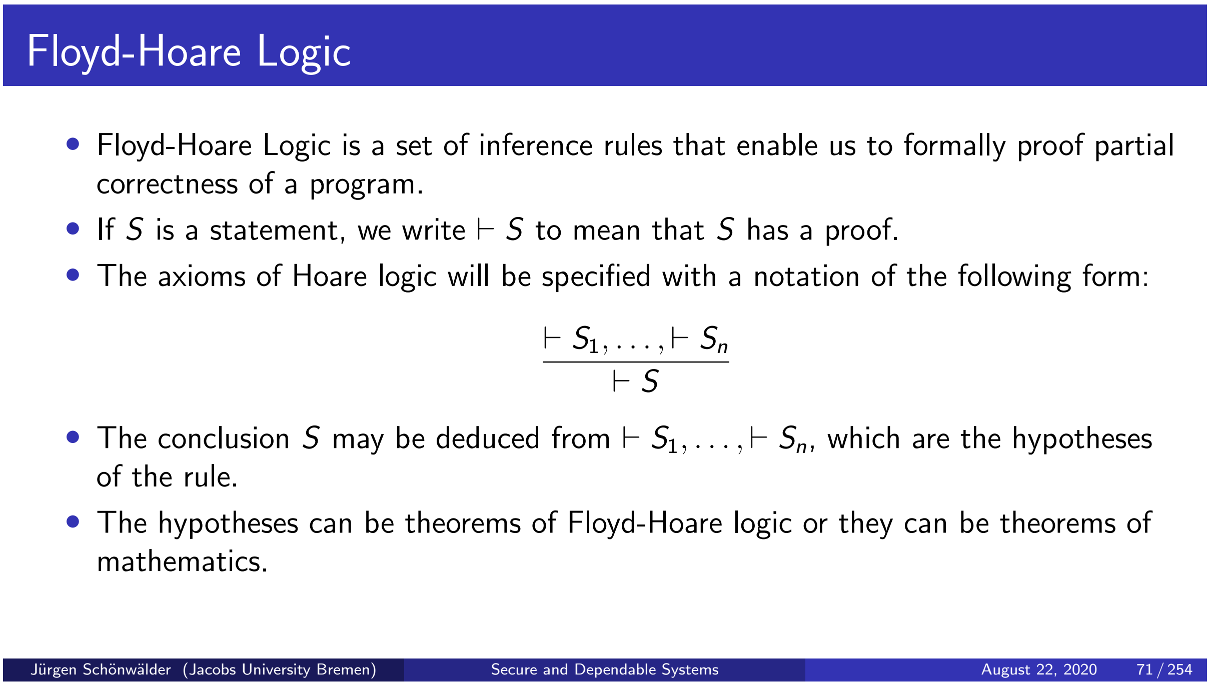
Software Verification

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:

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

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

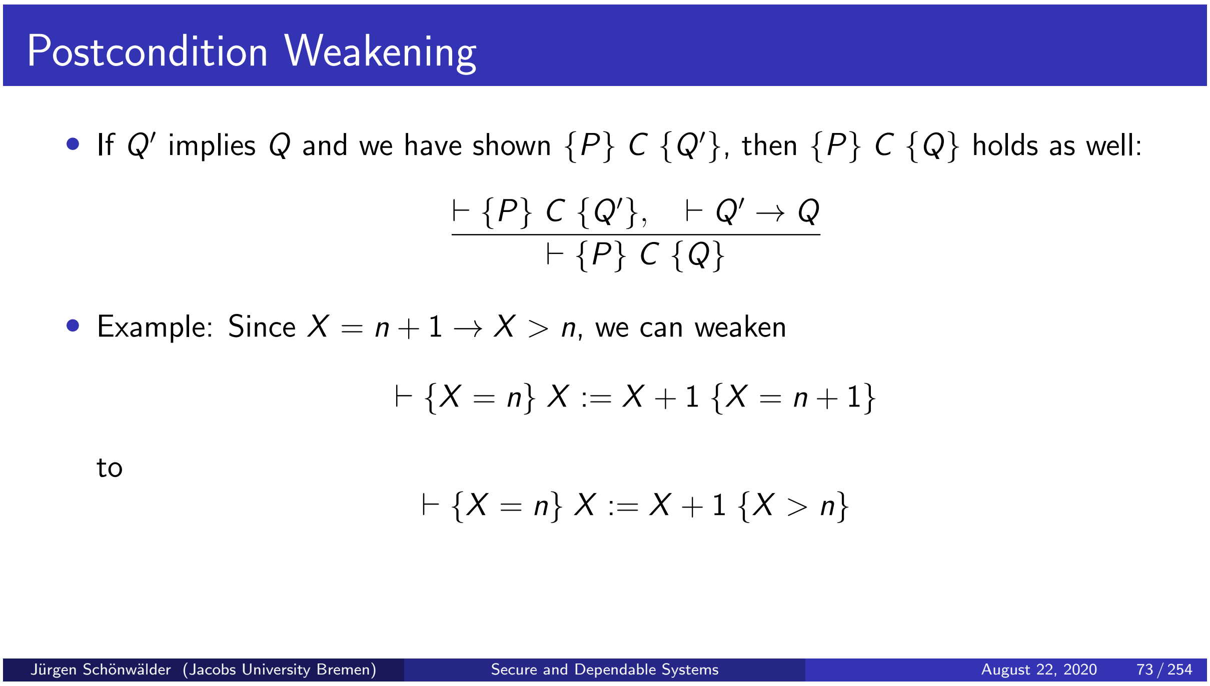
$$\vdash \{X + 1 = n + 1\}\ X := X + 1\ \{X = n + 1\}$$

to

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

The precondition P is stronger than P' ($P \to P'$) 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:

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

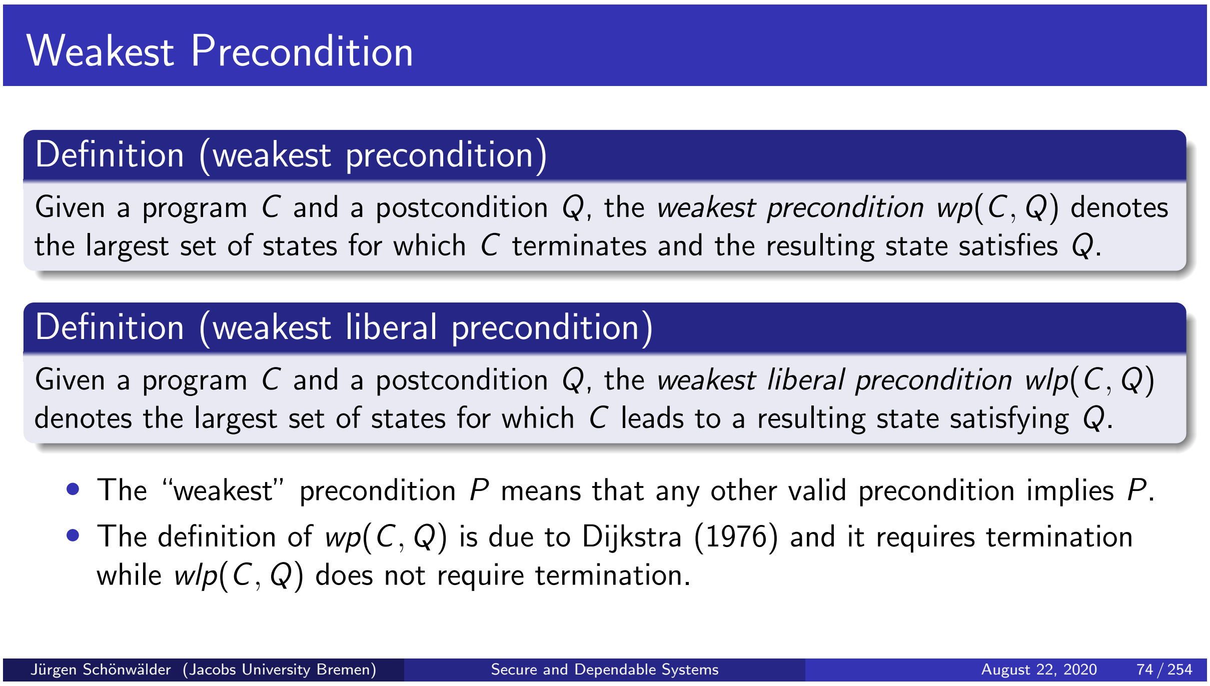
- Example: Since $X = n + 1 \rightarrow X > n$, we can weaken

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

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:

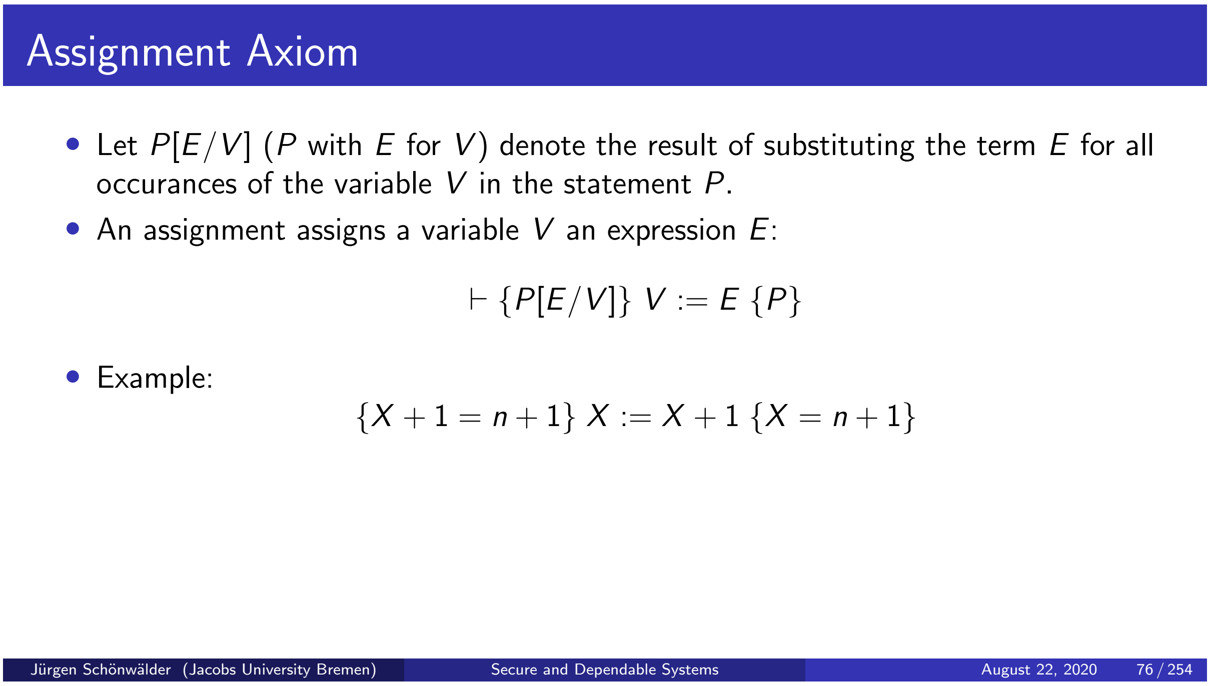
$$wp(X := X + 1, X > 0) = wlp(X := X + 1, X > 0) = (X > -1)$$

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:

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

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 76 / 254

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:

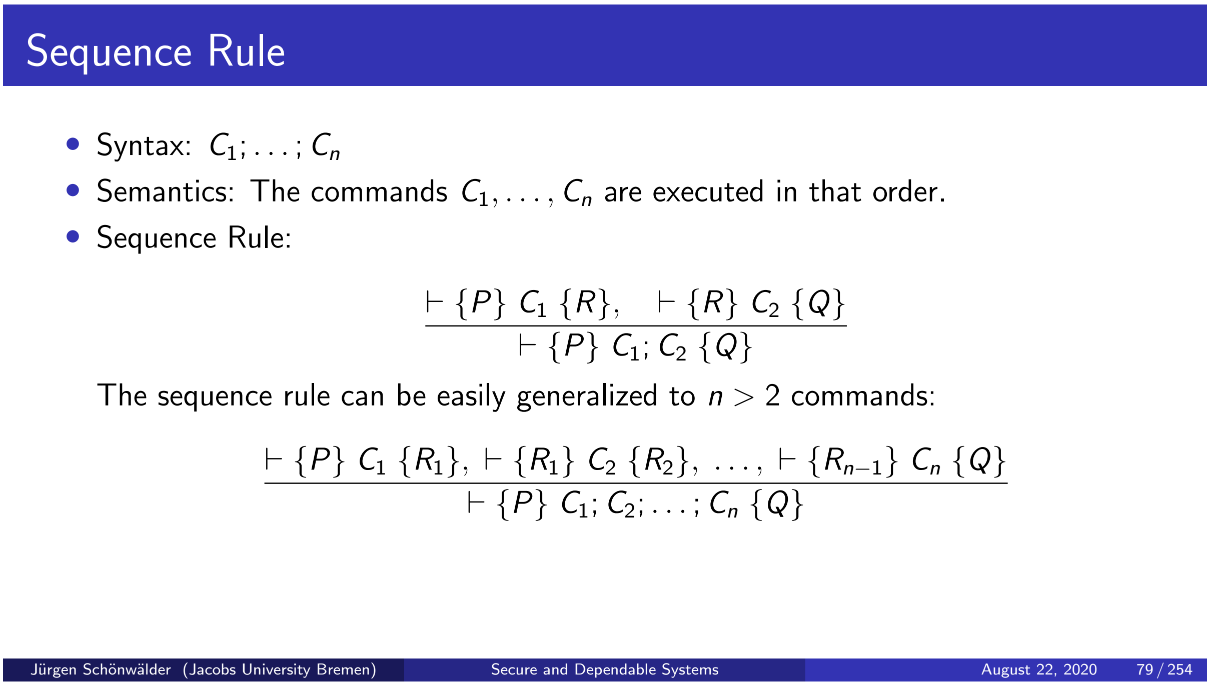
$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

- 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: $\{X = x \wedge Y = y\}$
$R := X$
$X := Y$
$Y := R$
Postcondition: $\{X = y \wedge Y = x\}$
```

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

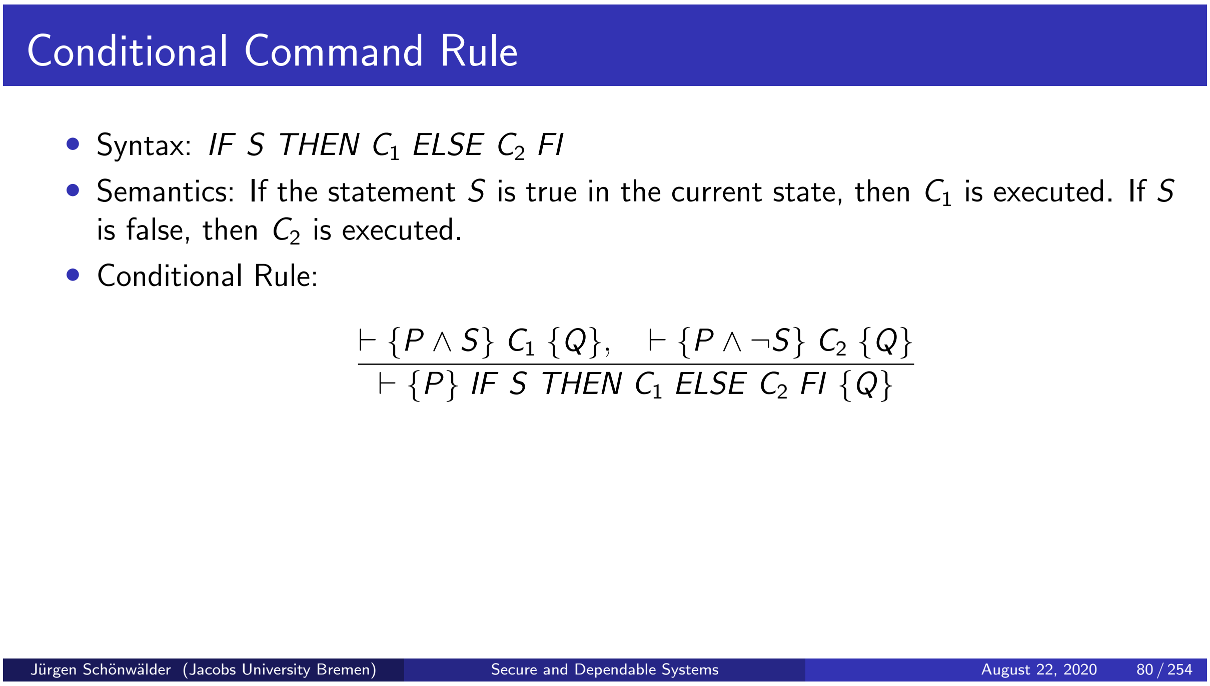
(i) $\vdash \{X = x \wedge Y = y\}\ R := X\ \{R = x \wedge Y = y\}$ (assignment axiom)

(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:

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

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 80 / 254

Consider the following specification and program (max):

```
Precondition: ${X = x \land Y = y}$
if $X \geq Y$ then
    $M := X$
else
    $M := Y$
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:

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

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

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:

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

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

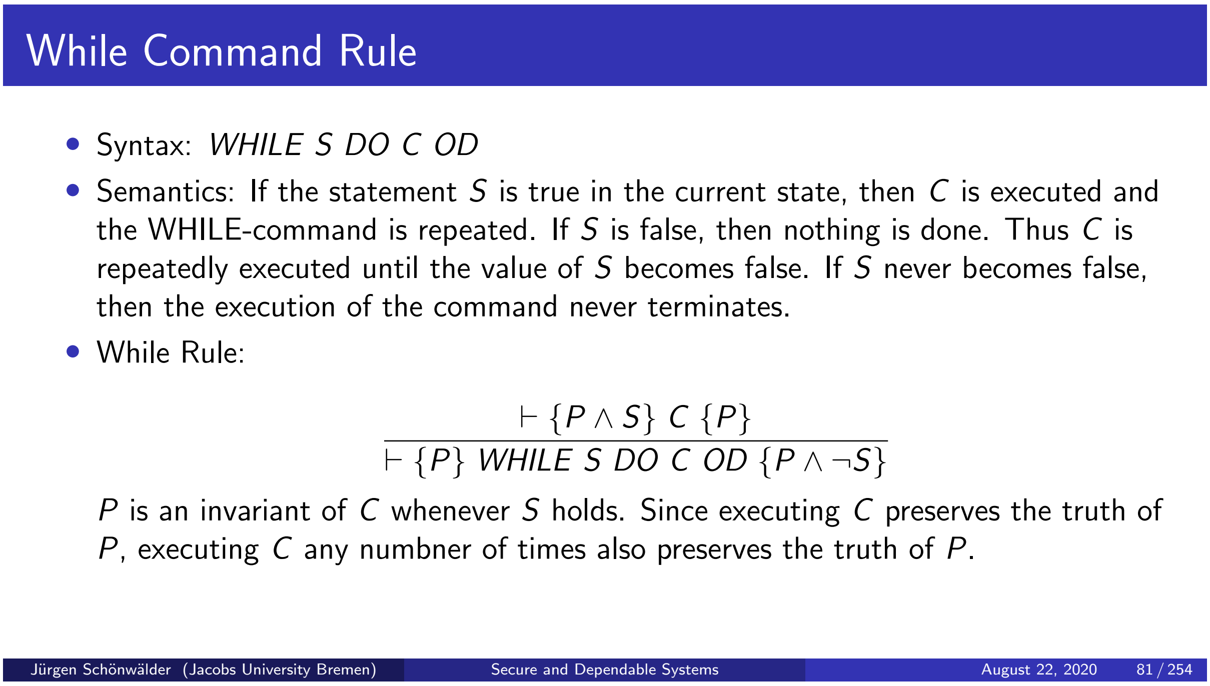
Postcondition weakening gives us:

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

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

Applying the conditional rule, we get:

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



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):

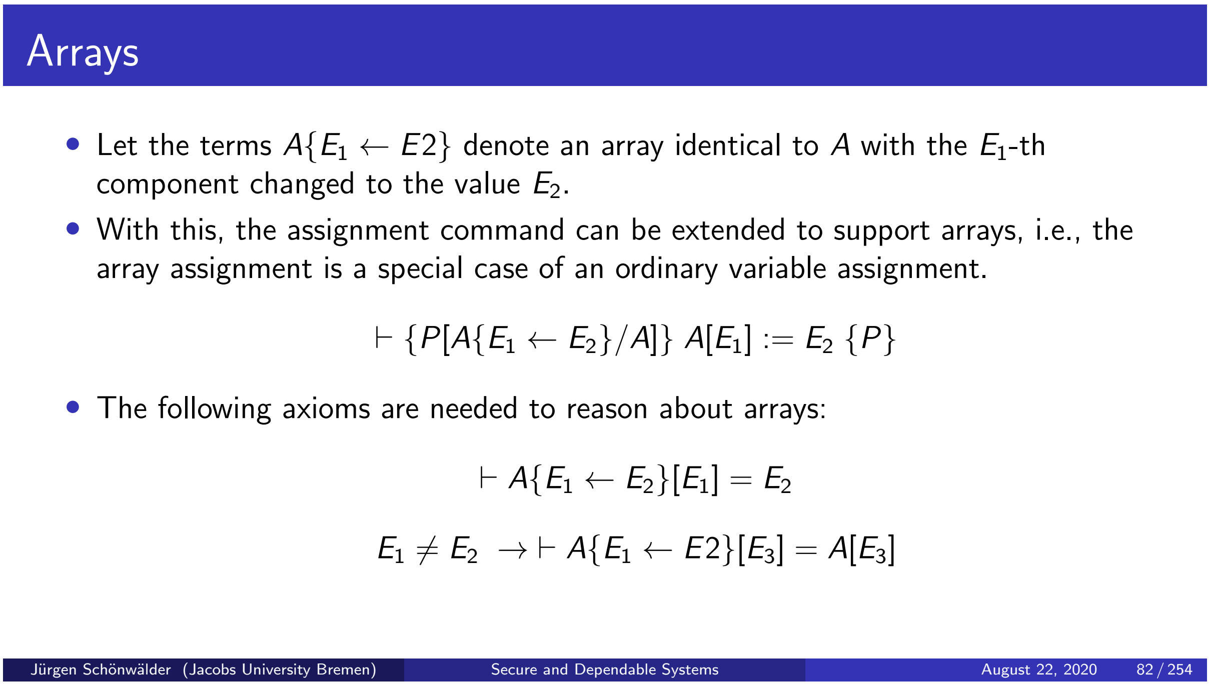
```
Precondition: $\{Y = 1 \wedge Z = 0 \wedge X = x \wedge X \geq 0\}$
while $Z \neq X$ do
    $Z := Z + 1$
    $Y := Y * Z$
od
Postcondition: $\{Y = x!\}$
```

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:

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

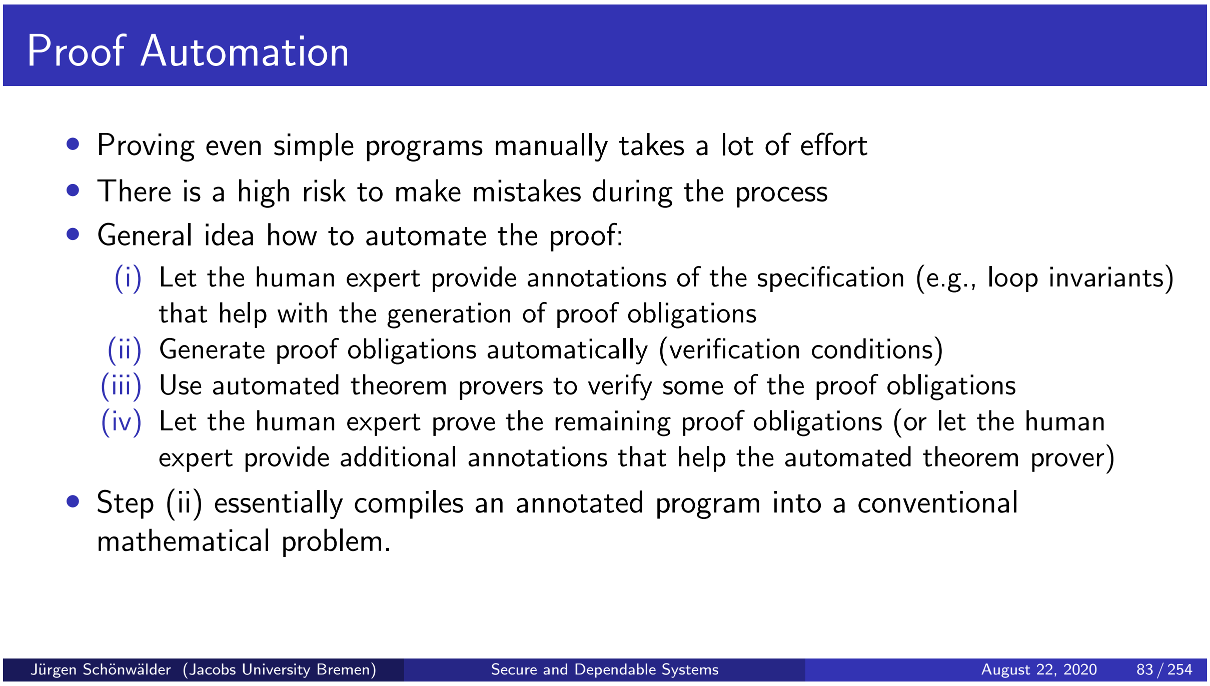
Applying the sequencing rule, we get:

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

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.

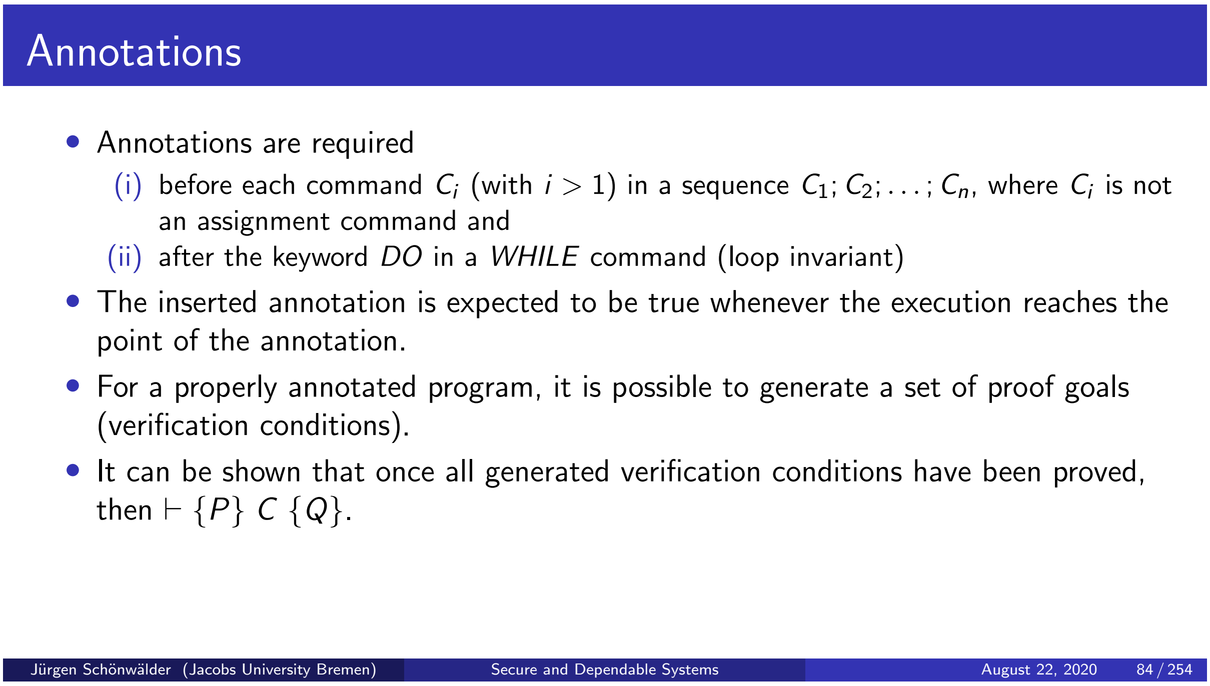


Proof Automation

- 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: ${\top}$
$R := X$
$Q := 0$
while $Y \leq R$ do
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \land R < Y\}$
```



Annotations

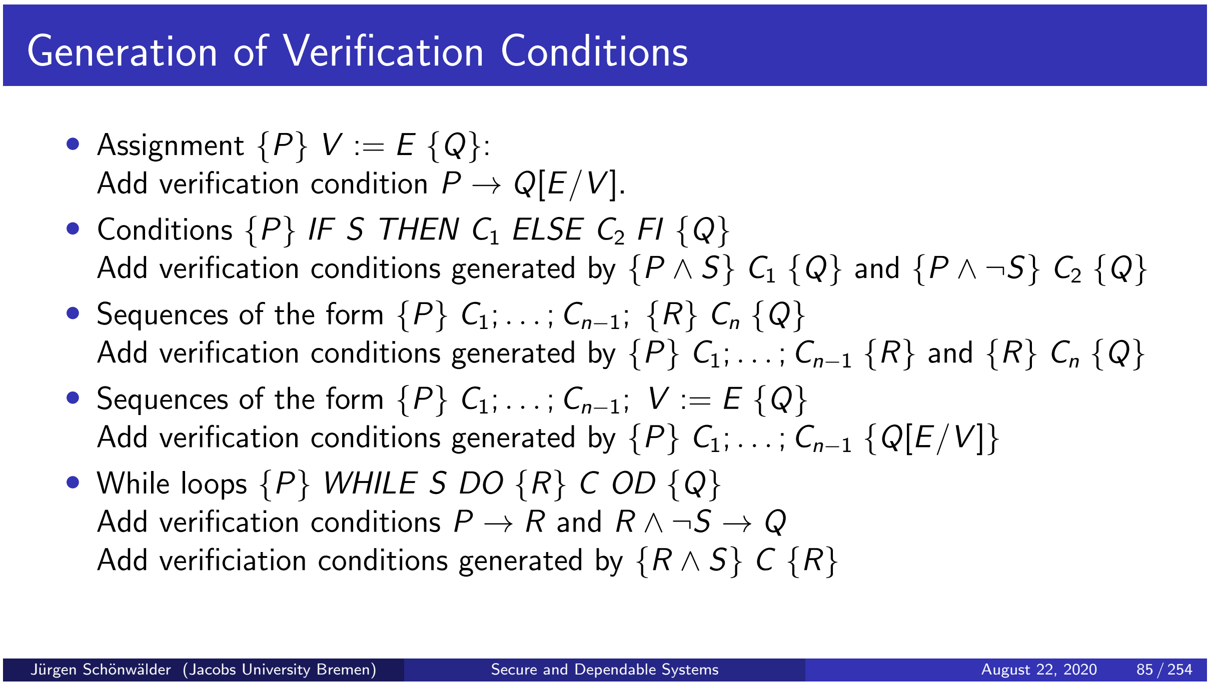
- Annotations are required
  (i) before each command C_i (with $i > 1$) in a sequence $C_1; C_2; \ldots; C_n$, where C_i is not an assignment command and
  (ii) after the keyword *DO* in a *WHILE* command (loop invariant)
- The inserted annotation is expected to be true whenever the execution reaches 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: $\{\top\}$
$R := X$
$Q := 0$
$\{R = X \wedge Q = 0\}$
while $Y \leq R$ do
    $\{X = Y \cdot Q + R\}$
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$
```

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

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



Generation of Verification Conditions

- Assignment $\{P\}\ V := E\ \{Q\}$:
  Add verification condition $P \rightarrow Q[E/V]$.
- Conditions $\{P\}$ *IF S THEN* C_1 *ELSE* C_2 *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 $\{P\}$ *WHILE S DO* $\{R\}$ *C OD* $\{Q\}$
  Add verification conditions $P \rightarrow R$ and $R \wedge \neg S \rightarrow Q$
  Add verificiation conditions generated by $\{R \wedge S\}\ C\ \{R\}$

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 85 / 254

Starting with the annotated example:

```
Precondition: $\{\top\}$
$R := X$
$Q := 0$
$\{R = X \wedge Q = 0\}$
while $Y \leq R$ do
    $\{X = Y \cdot Q + R\}$
    $R := R - Y$
    $Q := Q + 1$
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:

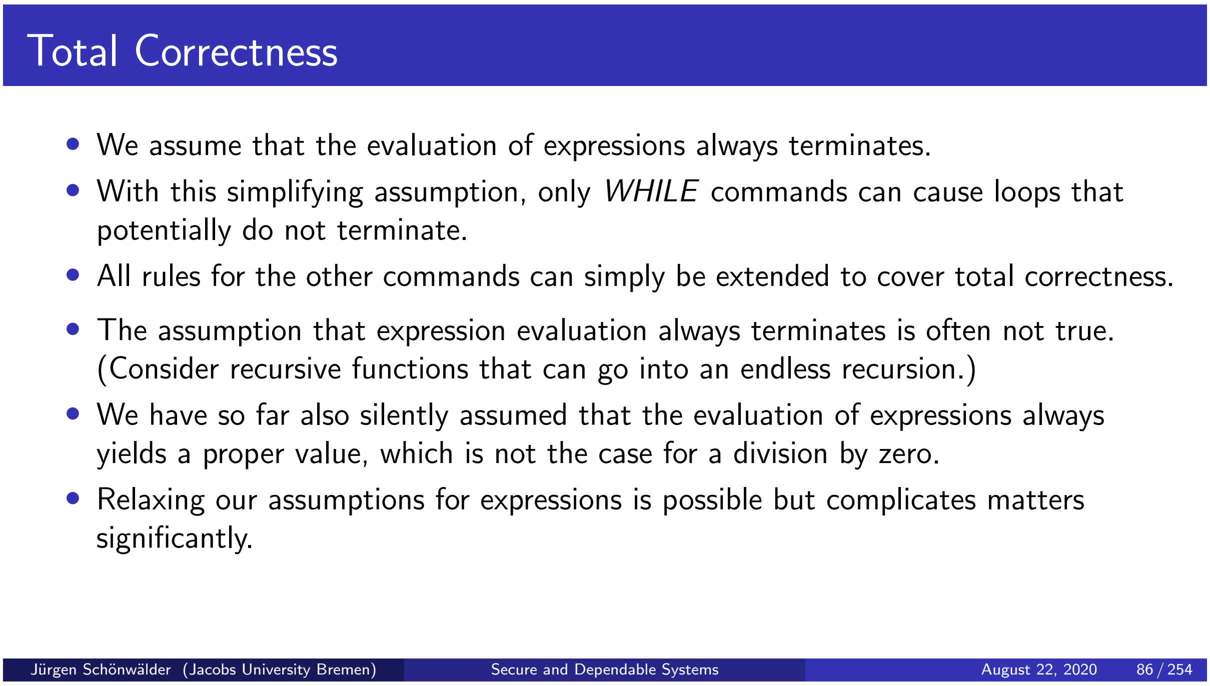
$$\{\top\}\ R := X;\ Q := 0\ \{R = X \wedge Q = 0\}$$
$$\{\top\}\ R := X\ \{R = X \wedge 0 = 0\}$$
$$\top \rightarrow (X = X \wedge 0 = 0)$$

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

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

- Specification disjunction

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

- Skip command rule

$$\frac{}{[P]\ \mathit{SKIP}\ [P]}$$

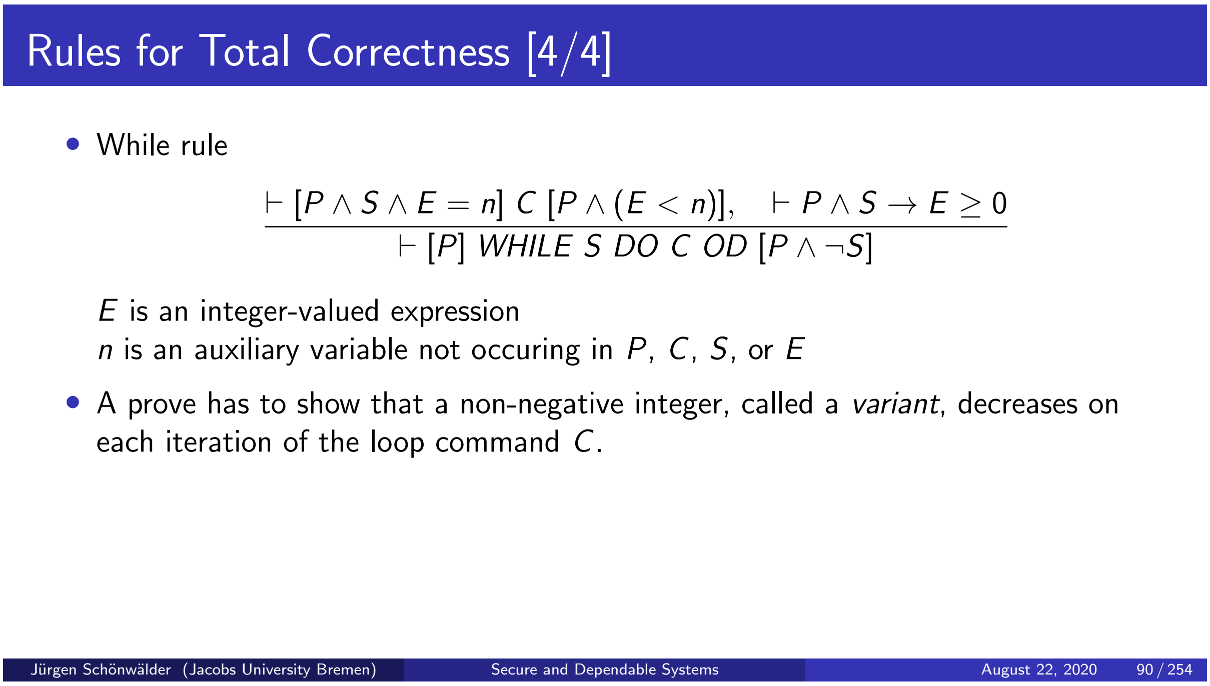
Rules for Total Correctness [3/4]

- 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

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

  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: $\{\top\}$
$R := X$
$Q := 0$
while $Y \leq R$ do
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \wedge R < Y\}$
```

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

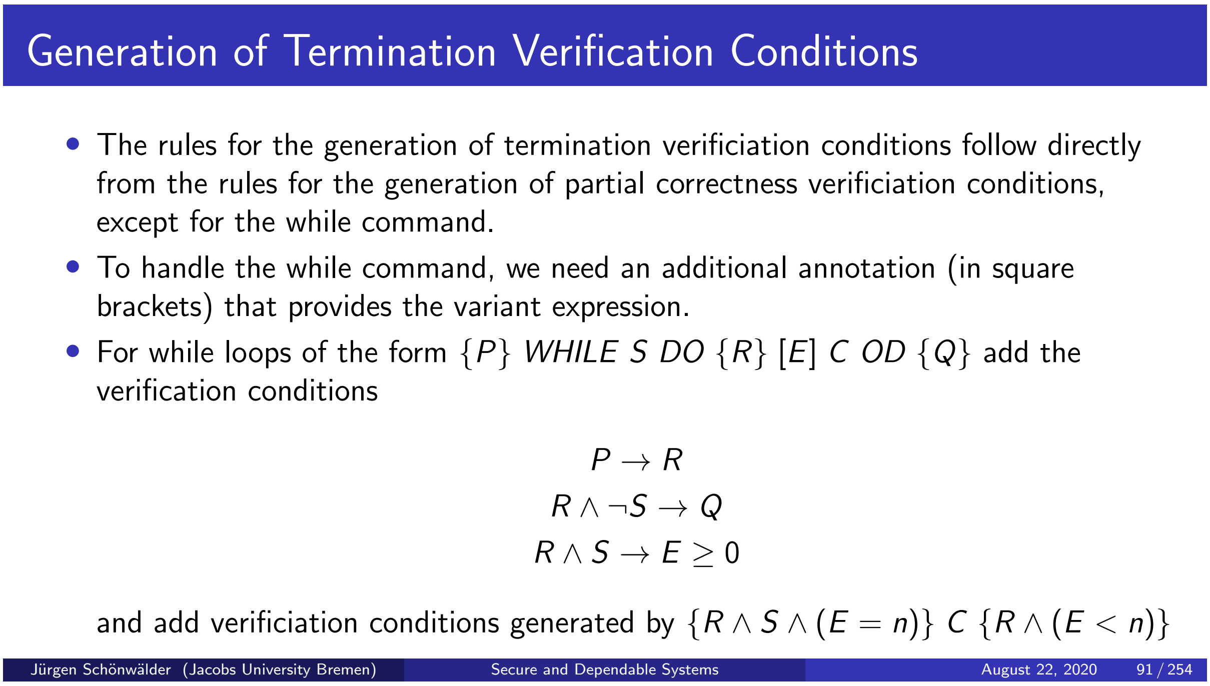
   This follows from the following derivation:

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

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

   This follows from:

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



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

$$P \to R$$
$$R \land \neg S \to Q$$
$$R \land S \to E \geq 0$$

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

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

```
Precondition: $\{\top\}$
$R := X$
$Q := 0$
$\{R = X \land Q = 0\}$
while $Y \leq R$ do
    $\{X = Y \cdot Q + R\}$
    $[R]$
    $R := R - Y$
    $Q := Q + 1$
od
Postcondition: $\{X = Y \cdot Q + R \land R < Y\}$
```

The while loop rule gives use the following termination VCs

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

and the VC generated as follows:

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

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

$Y = 0:$

$$((X = R \land 0 \leq R \land R = n) \to (X = R \land R < n)) \to \bot$$

$Y < 0:$

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

Termination and Correctness

- Partial correctness and termination implies total correctness:

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

- Total correctness implies partial correctness and termination:

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

Part III

Software Vulnerabilities and Exploits

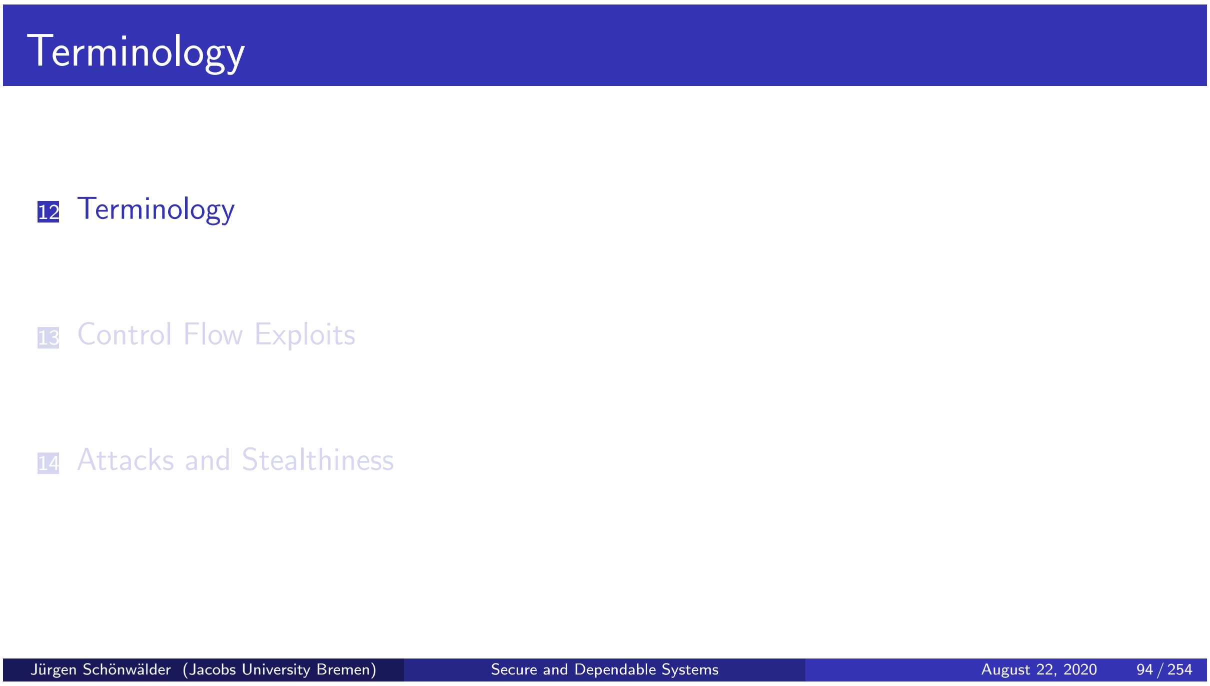
This part will introduce some basic techniques that can be used to attack systems. The main goal of an attack is typically to change the control flow of running programs and to make them execute arbitrary code provided by the attacker. While there are numerous attacks and exploits that have been disclosed (and likely more non-disclosed exploits), we will focus here on some classic techniques in order to develop an idea how the control flow of running programs can be changed.

Attacks on computing systems are often a sequence of steps:

1. Find vulnerabilies in software (or hardware).
2. Create an exploit for a vulnerability that ideally results in some control over a system.
3. If necessary, execute a privilege escalation attack to obtain more privileges
4. Install attack code that also tries to be invisible (stealthy)
5. Executing the attack

Note that the differnet steps involve different skills and are often done by different people. There is meanwhile an whole economy where organizations of attackers specialize on different activities and sell their provide services to others.

Note also that the first two steps are often substituted by social engineering techniques. In order to gain access to systems or data, it is often efficient to simply exploit human behavior. As computer scientist, we love to dive into deep technical aspects, often ignoring how simply it is to trick or persuade a human to gain access to computing systems.



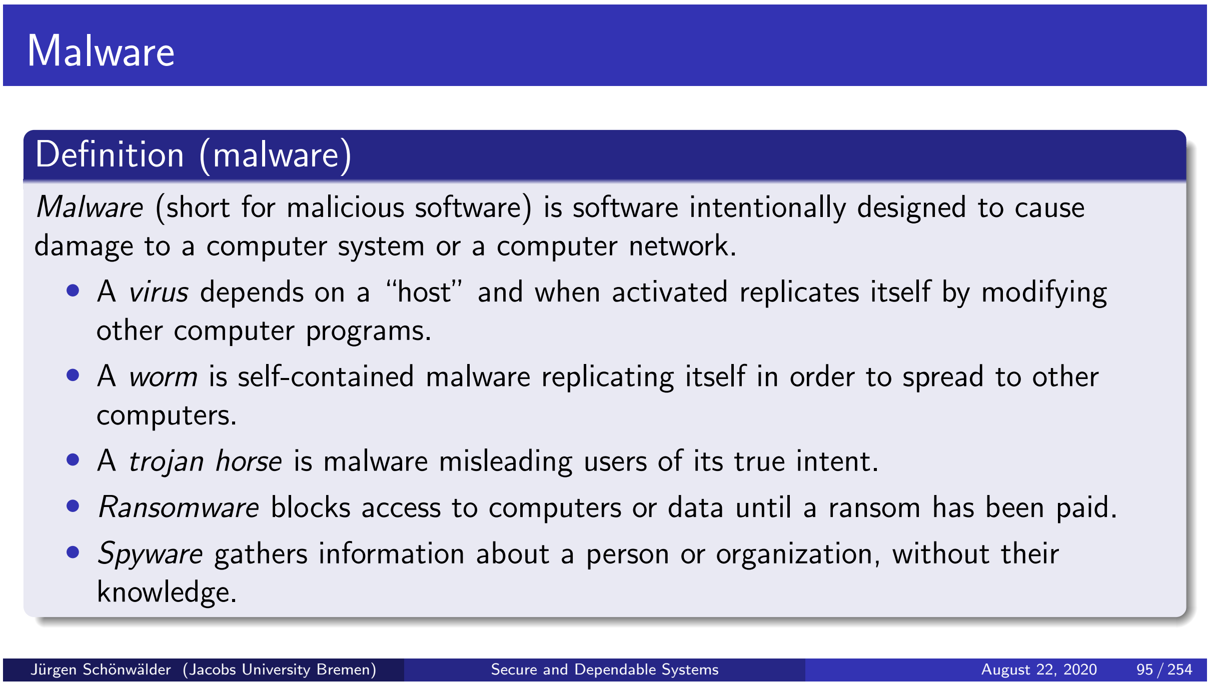
Terminology

12 Terminology

13 Control Flow Exploits

14 Attacks and Stealthiness

This section introduces some terminology commonly used in computer security. Our focus is on threats that can be exploited by an attacker for malicious purposes.



Malware

Definition (malware)

Malware (short for malicious software) is software intentionally designed to cause damage to a computer system or a computer network.

- A *virus* depends on a “host” and when activated replicates itself by modifying other computer programs.
- A *worm* is self-contained malware replicating itself in order to spread to other computers.
- A *trojan horse* is malware misleading users of its true intent.
- *Ransomware* blocks access to computers or data until a ransom has been paid.
- *Spyware* gathers information about a person or organization, without their knowledge.

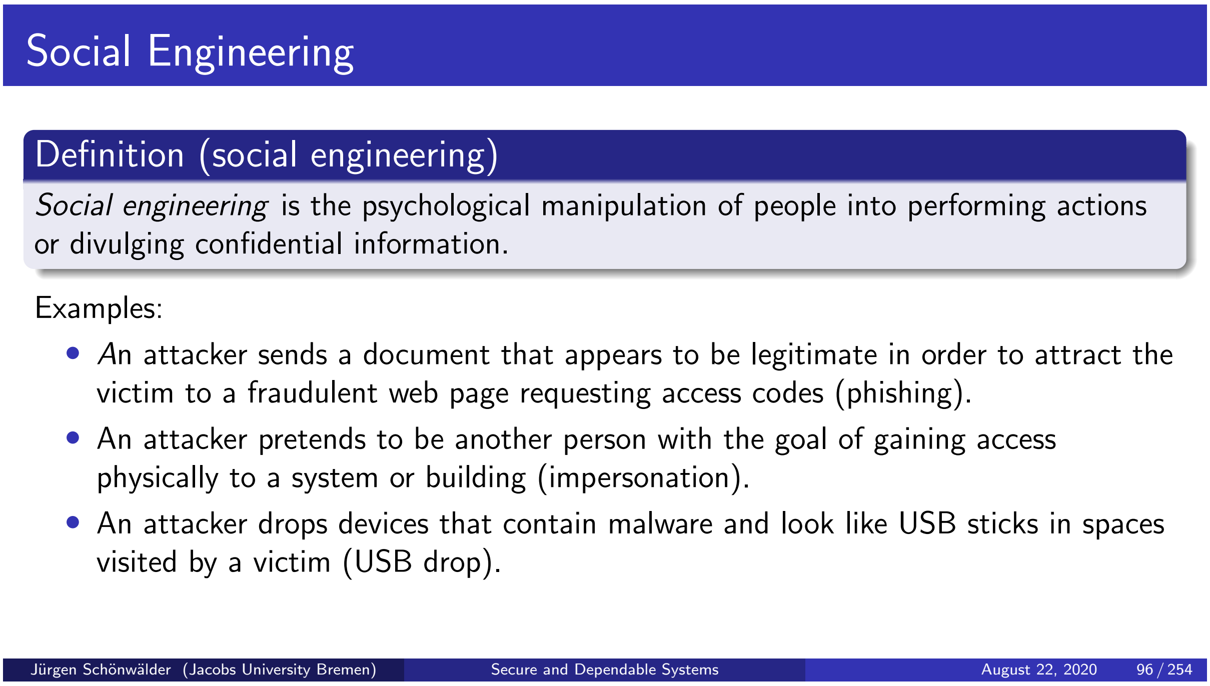
Viruses have been early forms of malware. They were often hiding in boot sectors of removable storage media such as floppy disks. Classic examples are “Pakistani Brain” and “Jerusalem”.

Worms often exploit communication networks to spread and hence they grew in popularity with the Internet. Some examples:

- Morris worm (aka Internet worm), 1988
- ILOVEYOU, May 2000
- Code Red, July 2001
- Sobig and Blaster worm, August 2003
- Sasser, May 2004

Trojan horses or short trojans hide their true intent. For example, a fun game loaded on a mobile device may not just be a game but also a program for collecting data or spying on users. Some examples:

- Back Orifice, 1998
- Gh0st RAT, 2009
- MiniPanzer and MegaPanzer, 2009
- Shedun, 2015



Social Engineering

Definition (social engineering)

Social engineering is the psychological manipulation of people into performing actions or divulging confidential information.

Examples:

- *A*n attacker sends a document that appears to be legitimate in order to attract the victim to a fraudulent web page requesting access codes (phishing).
- An attacker pretends to be another person with the goal of gaining access physically to a system or building (impersonation).
- An attacker drops devices that contain malware and look like USB sticks in spaces visited by a victim (USB drop).

Social engineering is very effective and often the cheapest way for an attacker to achieve his goals. Since social engineering attacks usually take time to prepare and carry out, they are often targeted to specific persons.

xkcd: Security

Backdoors

Definition (backdoor)

A *backdoor* is a method of bypassing normal authentication systems in order to gain access to a computer program and computing system. Backdoors might be created by malicious software developers, tools such as compilers, or by other forms of malware.

Examples:

- Well-known default passwords effectively function as backdoors.
- Debugging features used during development phases can act as backdoors.
- Backdoors may be inserted by a malicious compiler.

Rootkits

Definition (rootkit)

A *rootkit* . . .

Advanced Persistent Threats

Control Flow Exploits

12 Terminology

13 Control Flow Exploits

14 Attacks and Stealthiness

Intel's x86_32 processor architecture has eight general-purpose registers (eax, ebx, ecx, edx, ebp, esp, esi, edi). The x86_64 architecture extends them to 64 bits (prefix "r" instead of "e") and adds another eight registers (r8, r9, r10, r11, r12, r13, r14, r15). Some of x86 registers have special meanings and are not really used as general-purpose registers. The ebp (rbp) register is used to point to the beginning of a stack frame (base pointer)) while the esp (rsp) register is used to point to the top of the stack (stack pointer). (Note that the stack grows downwards on the x86 architecture.) There are additional special purpose registers, most important for us is the eip (rip) register, which points to the current instruction (instruction pointer).

Note that the base pointer ebp (rbp) is optional. It helps to debug programs but costs a few additional instructions on every function call.

In the following, we will only consider x86_64 processors. The stack frame layout using the common function calling conventions can be best explained with an example. Lets assume we have defined the following C function.

```c
long foo(long a, long b, long c, long d, long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = bar(xx, yy);
    return zz + 42;
}
```

The common function calling conventions will pass the first six arguments in the registers (rdi, rsi, rdx, rcx, r8, r9) and the remaining two arguments will be passed via the stack. The call instruction will then push the return address on the stack and the function prologue will push the old base register to the stack. The automatic function local variables xx, yy, and zz are then allocated on the stack as well by adjusting the stack pointer (rsp) accordingly.

Attacks and Stealthiness

12 Terminology

13 Control Flow Exploits

14 Attacks and Stealthiness

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.

That said, here is a quote from a book written by Kaufman, Perlman, and Speciner in 1995 [26], that still seems to be relevant:

> "Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations. They are also large, expensive to maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed. But they are sufficiently pervasive that we must design our protocols around their limitations."

Despite all weaknesses, passwords are still widely used to authenticate people. While there is recently a move towards two-factor authentication (e.g., a user needs to know something and to possess something), there is a long way to go until we have usable strong cryptographic security everywhere.

Another recent challenge is coming from the advances in quantum computing, which challenge some cryptographic techniques and require to invent new, so called quantum-resistant or post-quantum cryptographic algorithms.

Cryptography Primer

15 Cryptography Primer

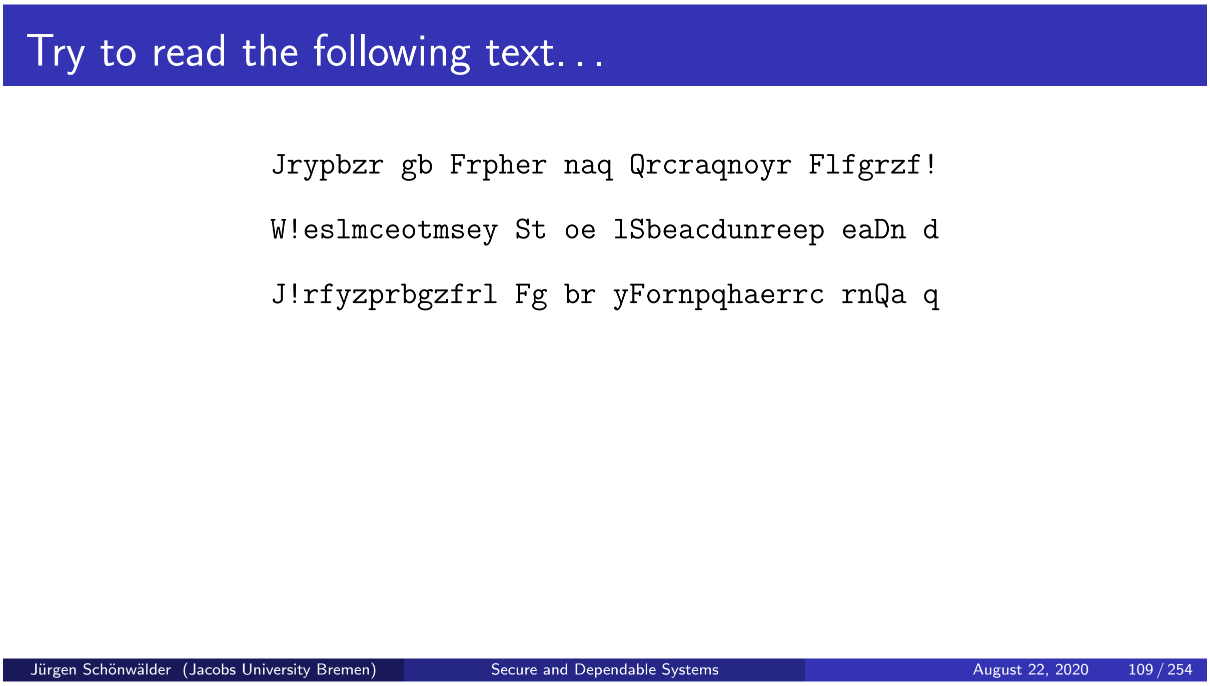
16 Symmetric Encryption Algorithms and Block Ciphers

17 Asymmetric Encryption Algorithms

18 Cryptographic Hash Functions

19 Digital Signatures and Certificates

20 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 rot_{13} algorithm, a simple letter substitution cipher that replaces a letter with the 13th letter after it, using the Latin alphabet and wrapping around as needed. Since the basic Latin alphabet has 26 letters, rot_{13} has the nice property that $m = rot_{13}(rot_{13}(m))$ for a given text m. The rot_{13} algorithm became popular in newsgroups of the 1980s in order to hide potentially offensive content. Applying rot_{13} 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:

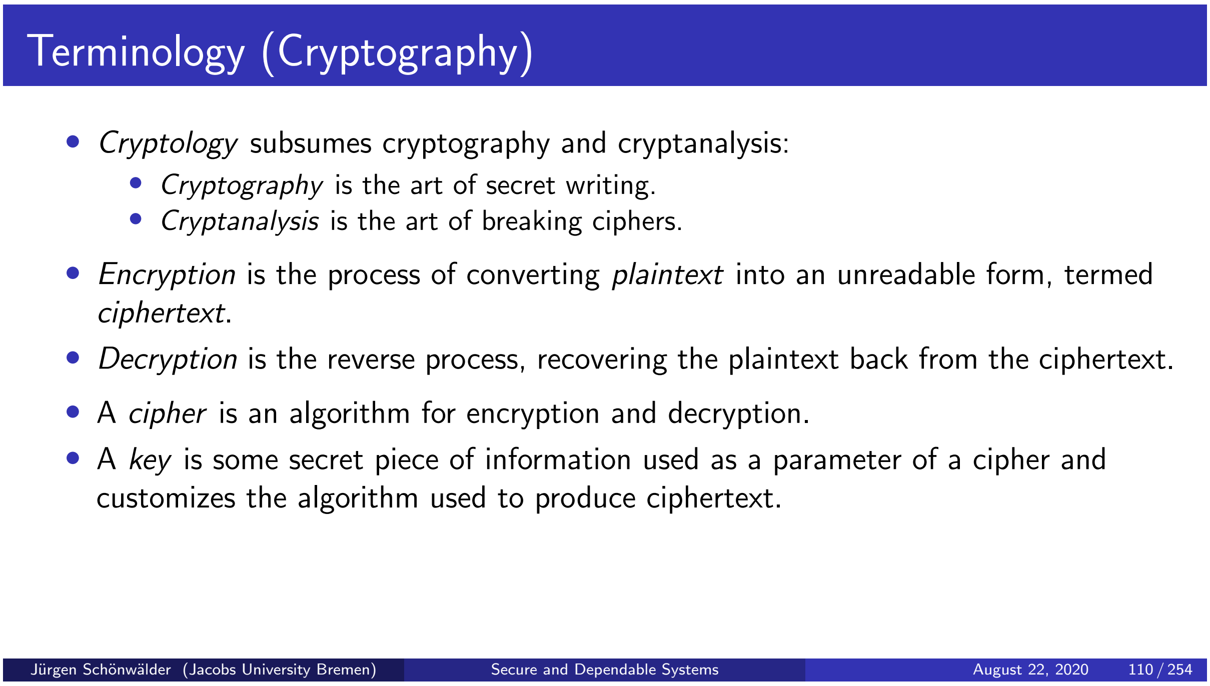
```
Welcome to Secure and Dependable Systems!
```

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!
```



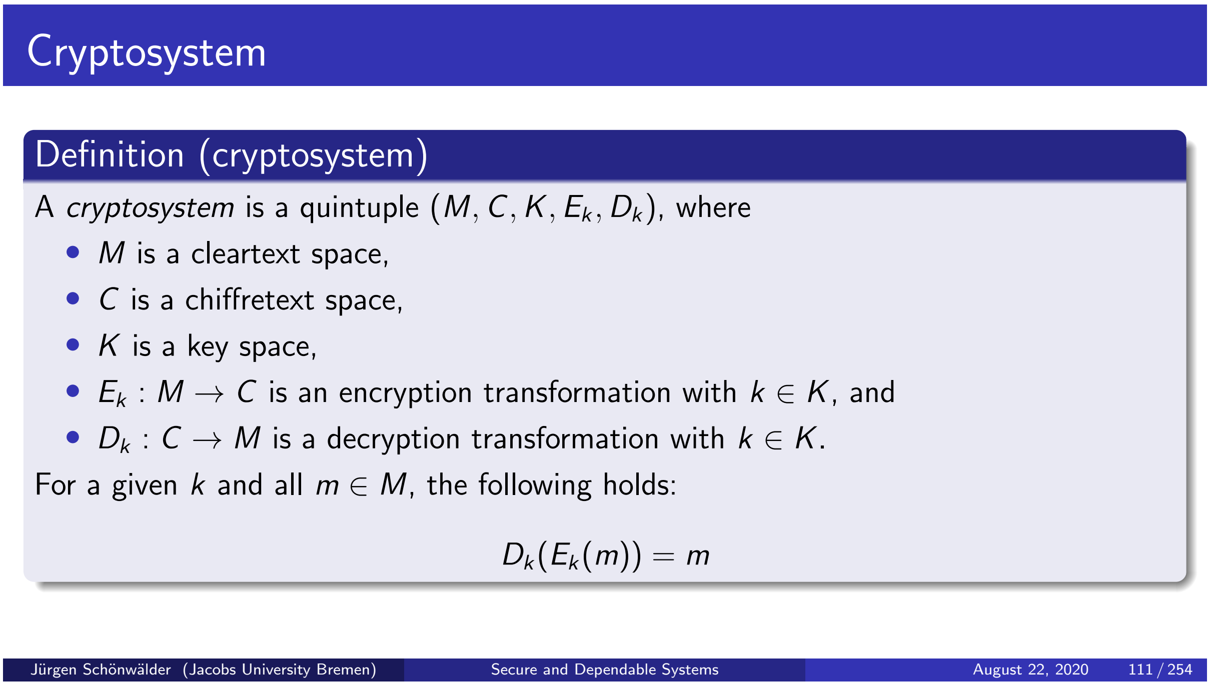
Terminology (Cryptography)

- *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 of the algorithms. The algorithms of good cryptosystems should be publically known and withstand any attempts to break them.

The following rules should be followed:

1. You should not keep your algorithm secret.
2. You do not know how much your algorithm is secret.
3. You cannot keep your algorithm secret.
4. But you can (and must) keep your password secret, and you can know and control ”how much” secret it is.



Cryptosystem

Definition (cryptosystem)

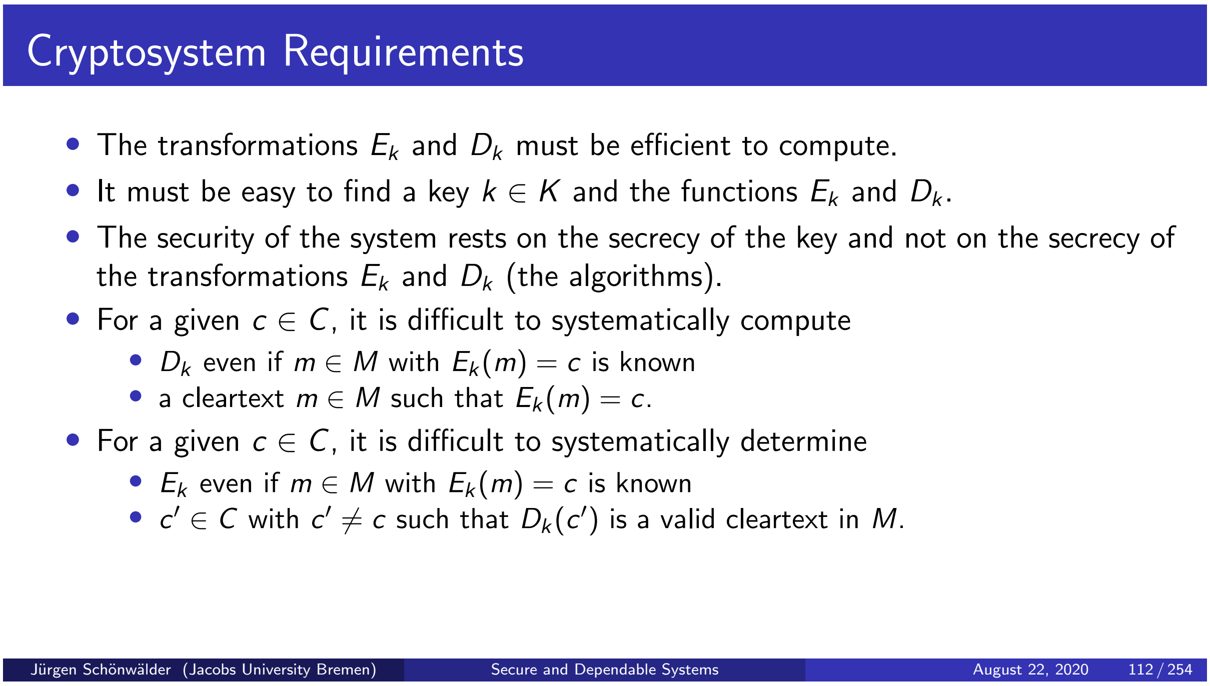
A *cryptosystem* is a quintuple (M, C, K, E_k, D_k), where

- 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:

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

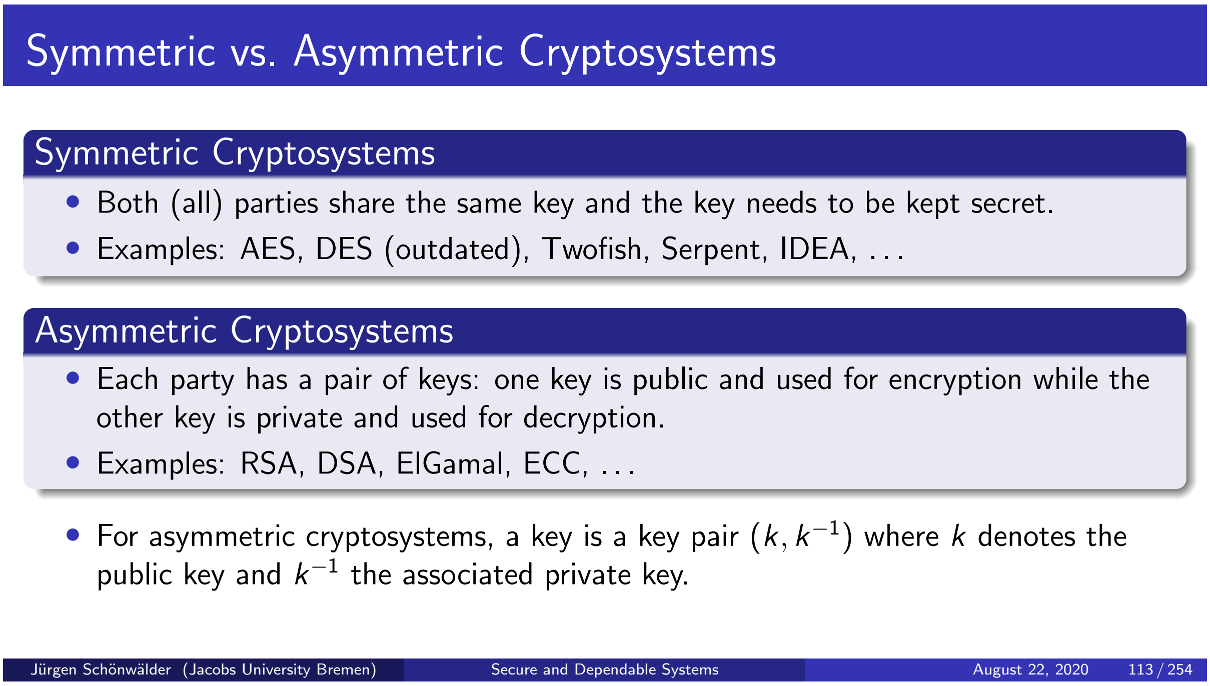
This definition is not yet complete. A cryptosystems must statisfy additional requirements since we do not want simple functions that are easy to revert.



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 E_k and D_k (the 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

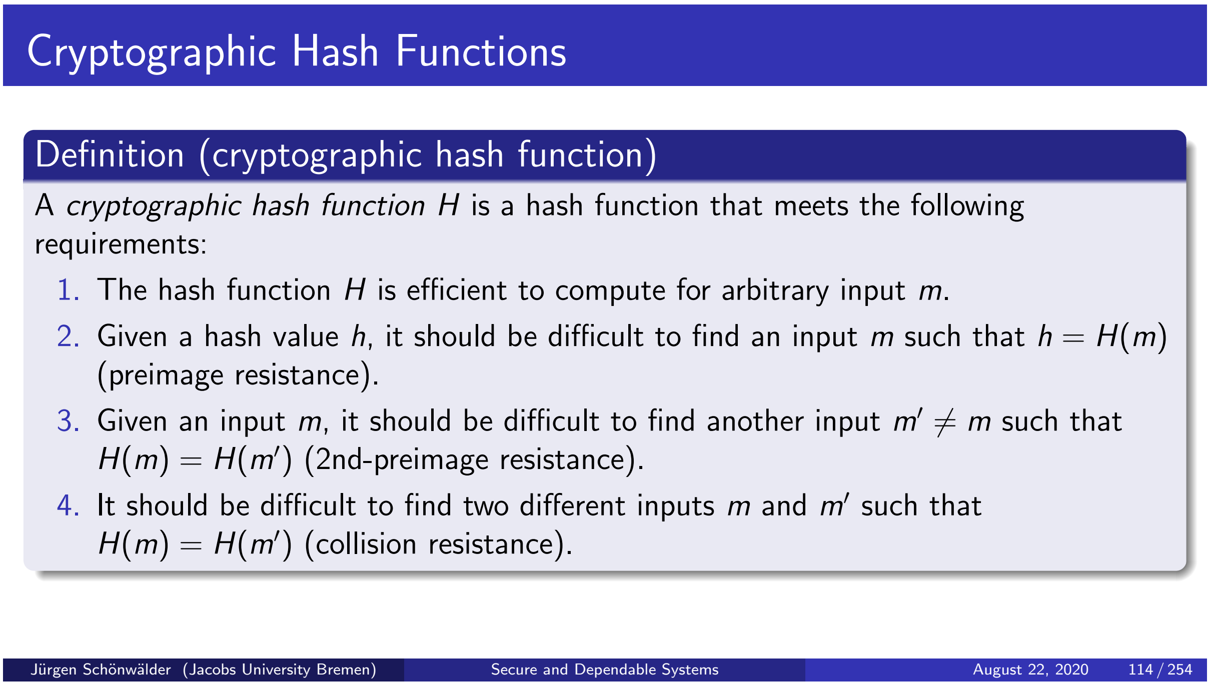
- Each party has a pair of keys: one key is public and used for encryption while the other key is private and used for decryption.
- Examples: RSA, DSA, ElGamal, ECC, ...

- 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 -pbkdf2 -in welcome.txt -out welcome.aes
$ openssl aes-256-cbc -pbkdf2 -d -in welcome.aes -out plaintext.txt
```

The commands will prompt you for a password, which is used to algorithmically derive the key that is used for encryption and decryption. (The option `-pbkdf2` tells `openssl` to use the password-based key derivation function 2.) The key itself is 256 bit long. We will discuss the different modes of operations like CBC soon.



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).

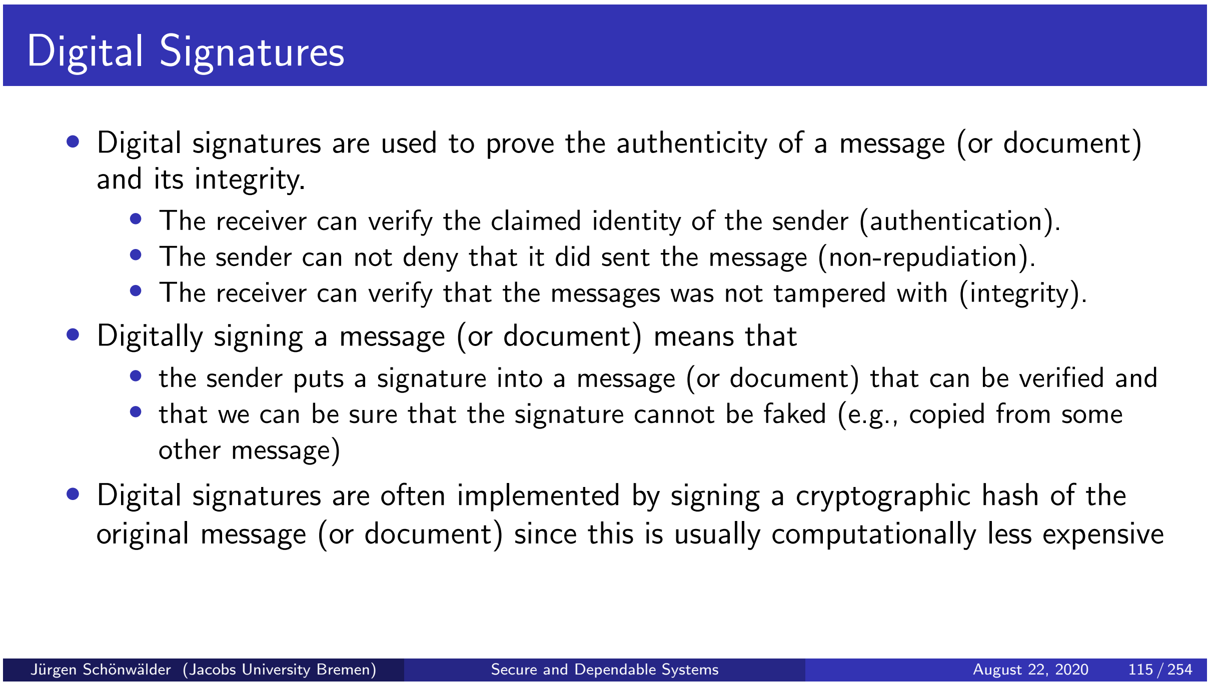
Cryptographic hash functions are used to compute a fixed size fingerprint, also called a message digest, of a variable length (cleartext) message.

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).

Cryptographic hashes can be used to verify the integrity of data. A filesystem, for example, may choose to store digests of file content in meta data in order to verify the integrity of file content. Hashes can also be used as short “handles” for longer documents. For example, instead of cryptographically signing a long document, you only sign the cryptographic hash of the document.

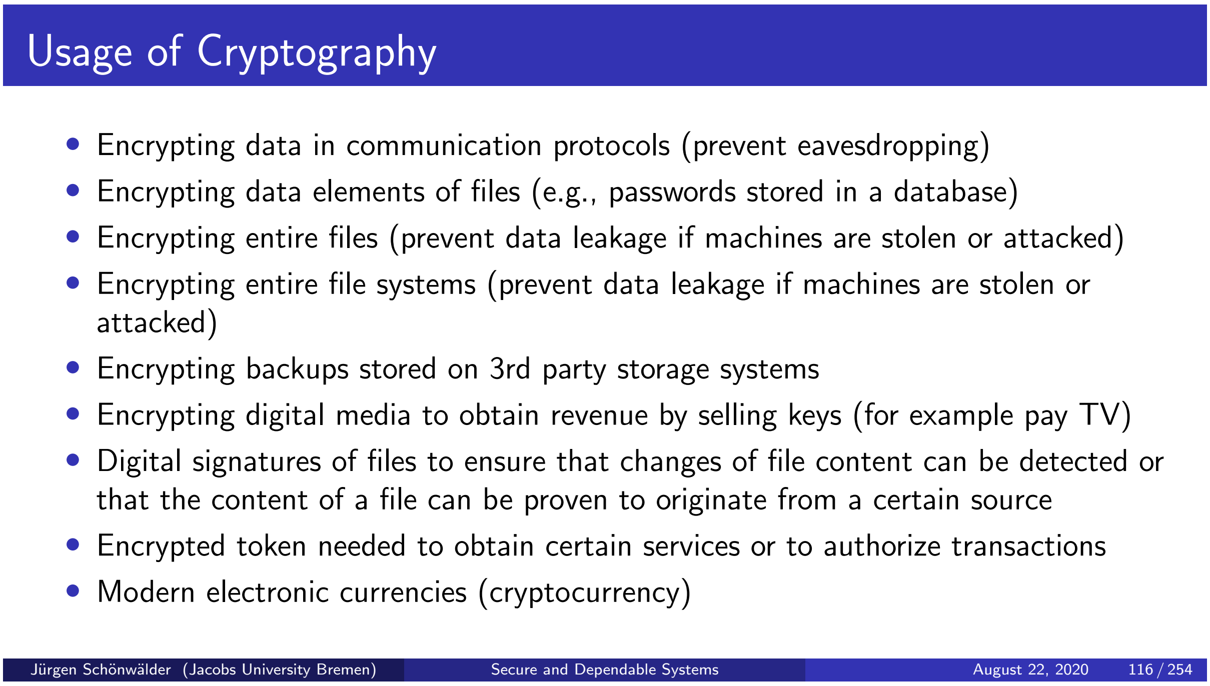


Digital Signatures

- 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 (authentication).
  - The sender can not deny that it did sent the message (non-repudiation).
  - The receiver can verify that the messages was not tampered with (integrity).
- 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 computationally less 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 [23] can be used to sign arbitrary digital artefacts. A JSON signature format is defined in RFC 7515 [24] and a CBOR signature format in RFC 8152 [44].

In the traditional analog world, we often sign documents using a handwritten signature. If the document is long, we often do not sign each page but instead we may flip a corner and sign over the flipped corner. The security of the system depends on the verification of handwritten signatures (which often can be forged with some amount of training). Digital signatures are actually stronger than many analog signatures since they allow to verify the integrity of the entire document. A downside of digital signatures is that they rely on digital keys that can reasonably be trusted.



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. Computer scientists and developers 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.

Cryptocurrencies like bitcoin and the underlying technology of blockchains have received much attention in the years 2016 and 2017. They are largely just another clever usage of cryptographic hash functions.

Symmetric Encryption Algorithms and Block Ciphers

15 Cryptography Primer

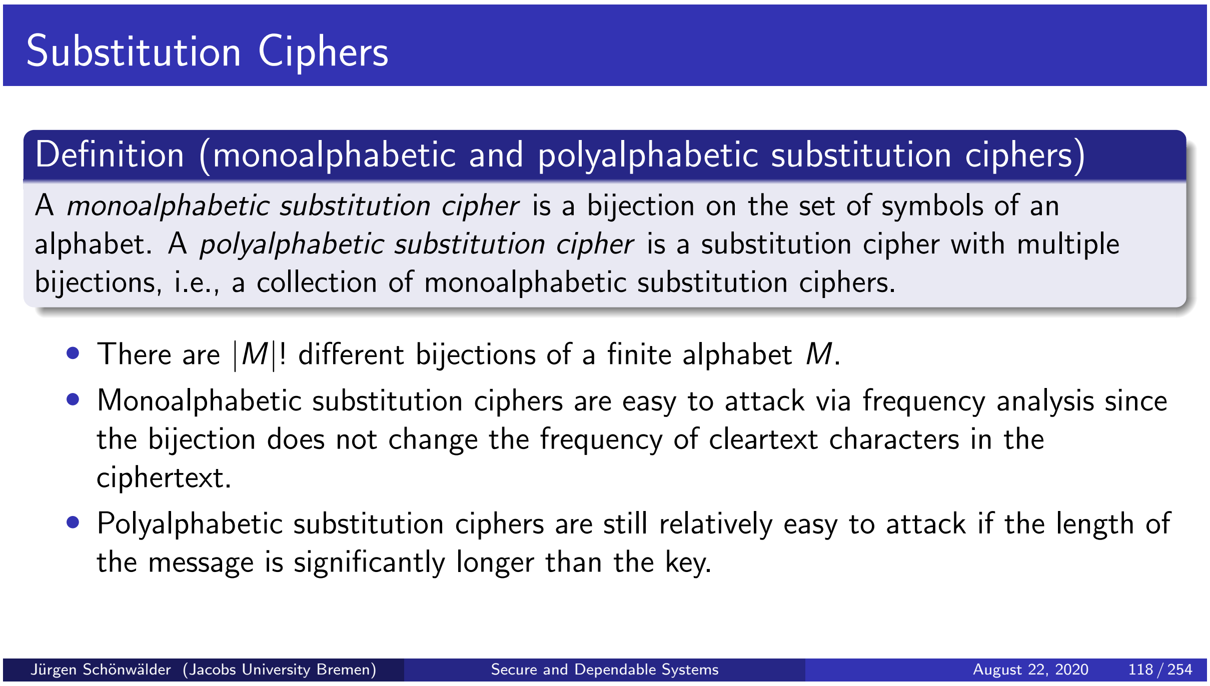
16 Symmetric Encryption Algorithms and Block Ciphers

17 Asymmetric Encryption Algorithms

18 Cryptographic Hash Functions

19 Digital Signatures and Certificates

20 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 rot_{13} 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).

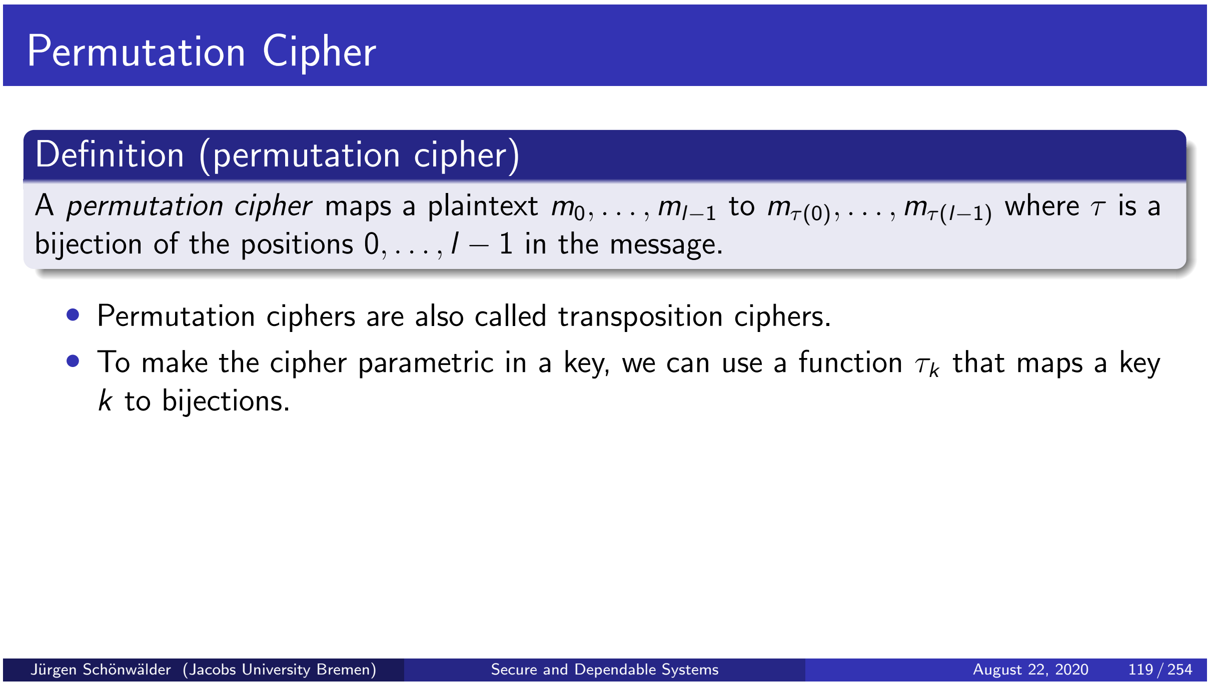
Lets represent all data as a number in $\mathbb{Z}_n$ (e.g., using ASCII code points or Unicode code points). Then we can consider monoalphabetic cryptosystems $(M = \mathbb{Z}_n, C = \mathbb{Z}_n, K = \mathbb{Z}^l, E_k, D_k)$ with

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

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. If the key is truly random, at least as long as the plaintext, never reused in whole or in part, and kept completely secret, then the resulting ciphertext will be impossible to decrypt or break.



Permutation Cipher

Definition (permutation cipher)

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

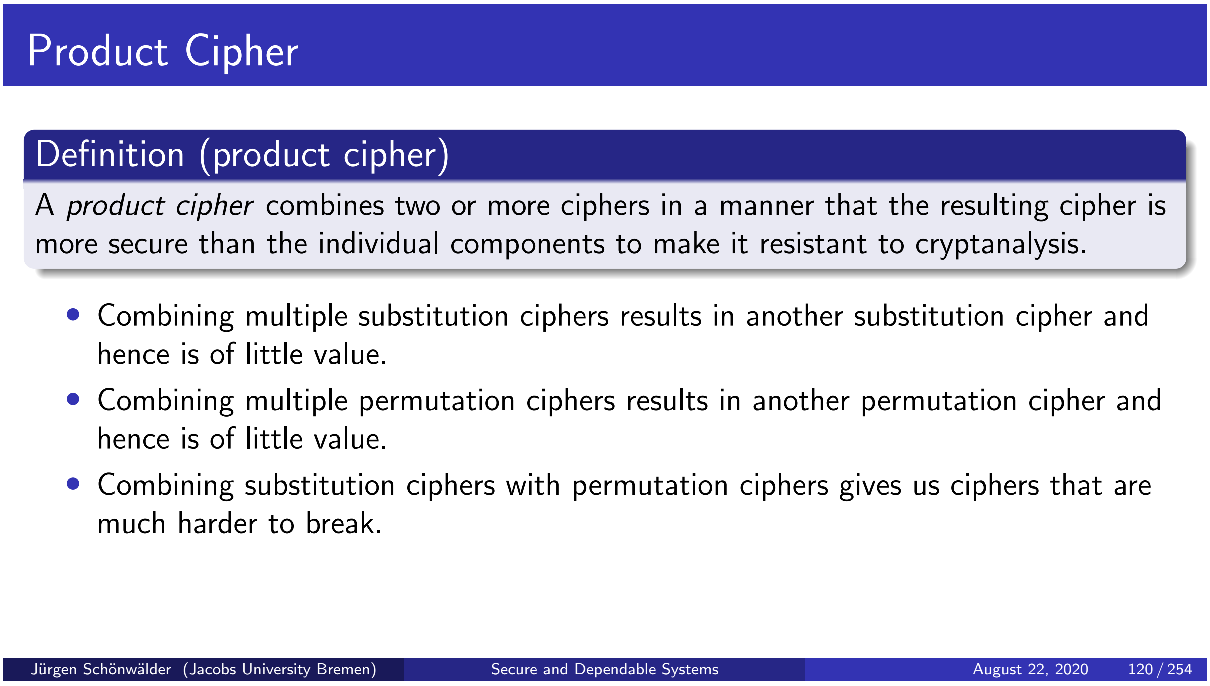
- Permutation ciphers are also called transposition ciphers.
- To make the cipher parametric in a key, we can use a function τ_k that maps a key k to bijections.

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.

Product ciphers are very important as they are the common foundation of many symmetric cipher algorithms. A specific class of product ciphers are so Feistel ciphers, named after the physicist and cryptographer Horst Feistel. Feistel ciphers work in rounds and in every round a key k_i is used. The sequence of keys k_i is typically generated from a key k using a key generator. In addition, in every round a round function F is applied.

A Feistel cipher encrypts data as follows:

1. Split the cleartext m into two equal size piezes l_0 and r_0

$$(l_0||r_0) = m$$

2. For each round $i = 0, 1, \ldots, n$ compute

$$l_{i+i} = r_i$$

$$r_{i+1} = l_i \oplus F(r_i, k_i)$$

The decryption works in a similar way backwards:

1. For each round $i = n, n-1, \ldots, 0$ compute

$$r_i = l_{i+1}$$

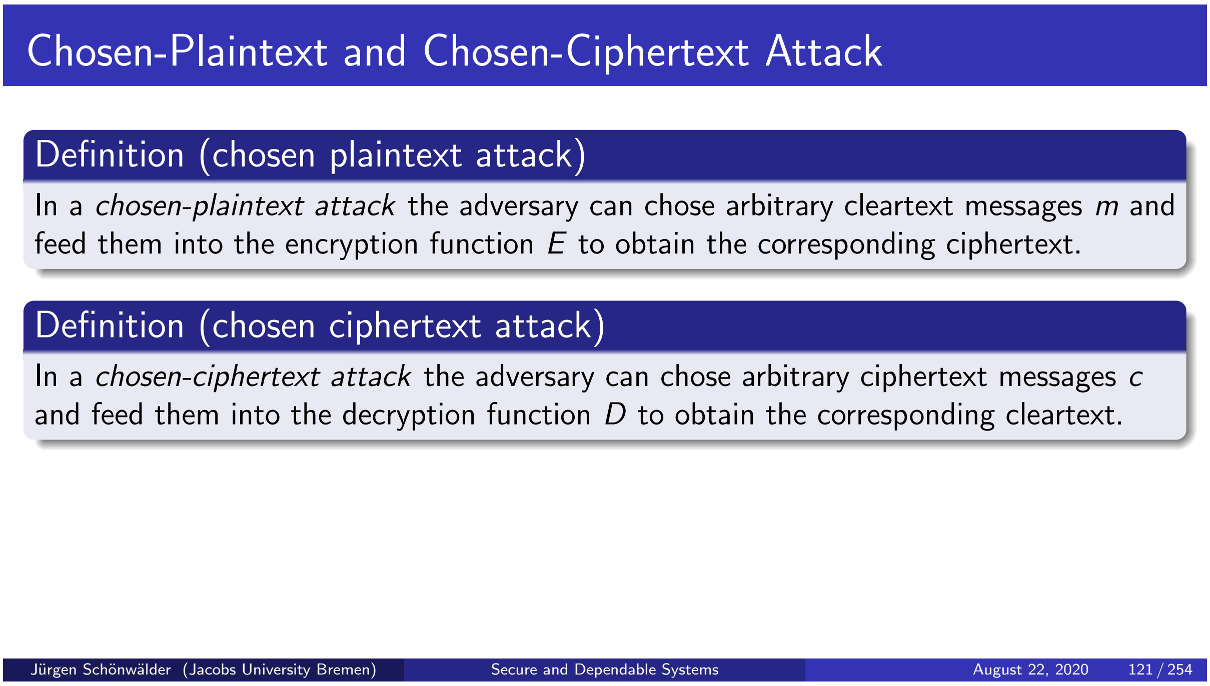
$$l_i = r_{i+1} \oplus F(l_{i+1}, k_i)$$

2. The plaintext m is obtained by concatenating l_0 and r_0

$$m = (l_0||r_0)$$

An interesting property of Feistel ciphers is that the function F does not have to be invertable. This means that, for example, a cryptographic hash function can be used as F.

YouTube: Feistel Cipher



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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 121 / 254

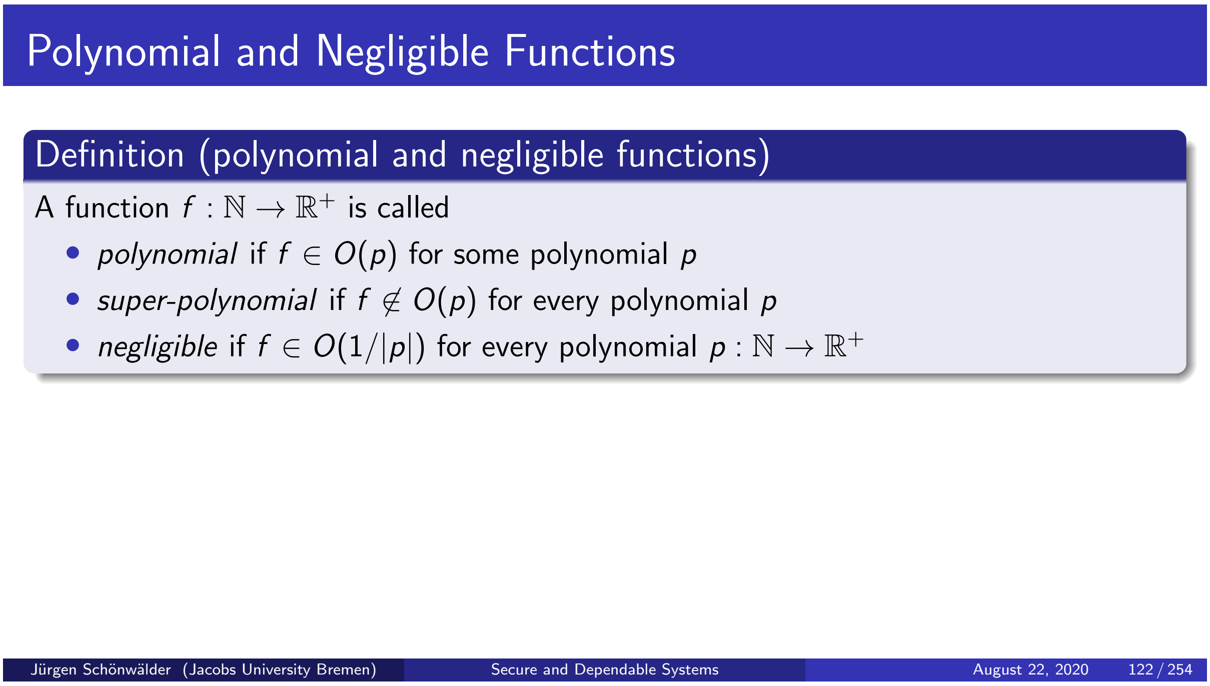
Attacks on cryptographic algorithms become easier if it is possible to “play” with the encryption or decryption functions. For example, feeding the characters

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
AAABBBAAA
```

into rot_{13} gives us

```
NOPQRSTUVWXYZABCDEFGHIJKLM
NNNOOONNN
```

which quickly gives us the idea that this is a simple substitution.



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 $p : \mathbb{N} \to \mathbb{R}^+$

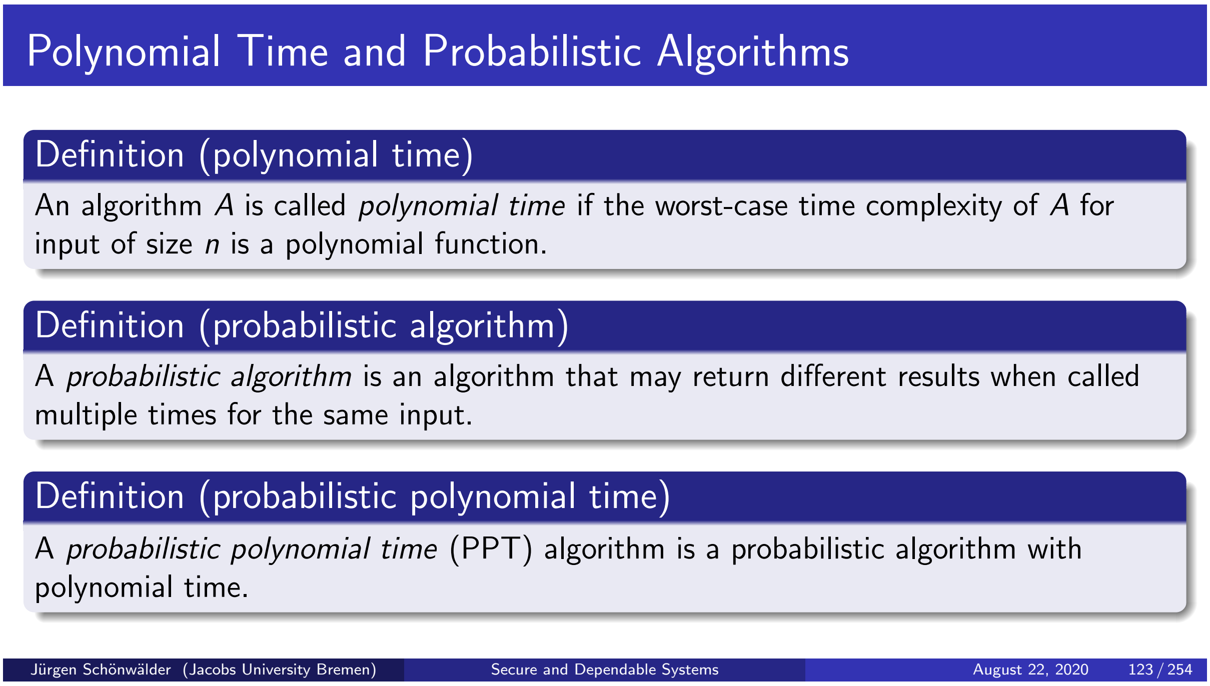
In modern cryptography, a security scheme is provably secure if the probability of security failure is negligible in terms of the cryptographic key length n.

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.

Examples:

$f(n) = x^{-n}$ is negligible for any $x \geq 2$



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. But we can use this property to decide whether a number is composite.

```
Require: $n > 3, k > 0$
while $k \neq 0$ do
    $x \leftarrow random(2, n-2)$                    ▷ pick x randomly in $[2, n-2]$
    if $x^{(n-1)} \not\equiv 1 \pmod{n}$ then
        return 0                                     ▷ composite
    fi
    $k \leftarrow k - 1$
end
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 $\mathbb{Z}_n$ are irrelevant because we take them modulo n anyway.
- The values $x = 0$ and $x = 1$ are useless because the property anyway holds for them.
- The value $x = n - 1$ is useless because the property holds anyway if n is odd.

Probabilistic prime number tests are of practical importance for traditional cryptographic algorithms that require large prime numbers in order to generate suitable keys. The idea is to find quickly possible candidates of prime numbers before verifying with a slower algorithm the prime number property. There are more advanced algorithms, most notably perhaps the Miller-Rabin primality test.

One-way Functions

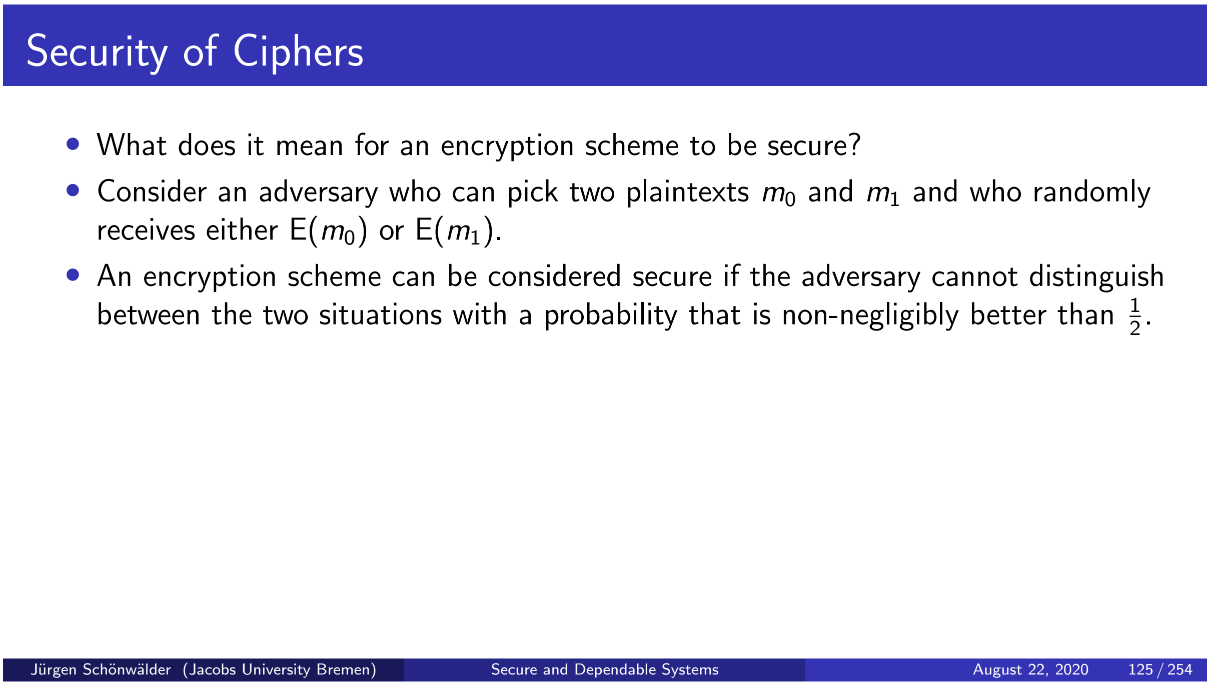
Definition (one-way function)

A function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ 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 factorization algorithm.

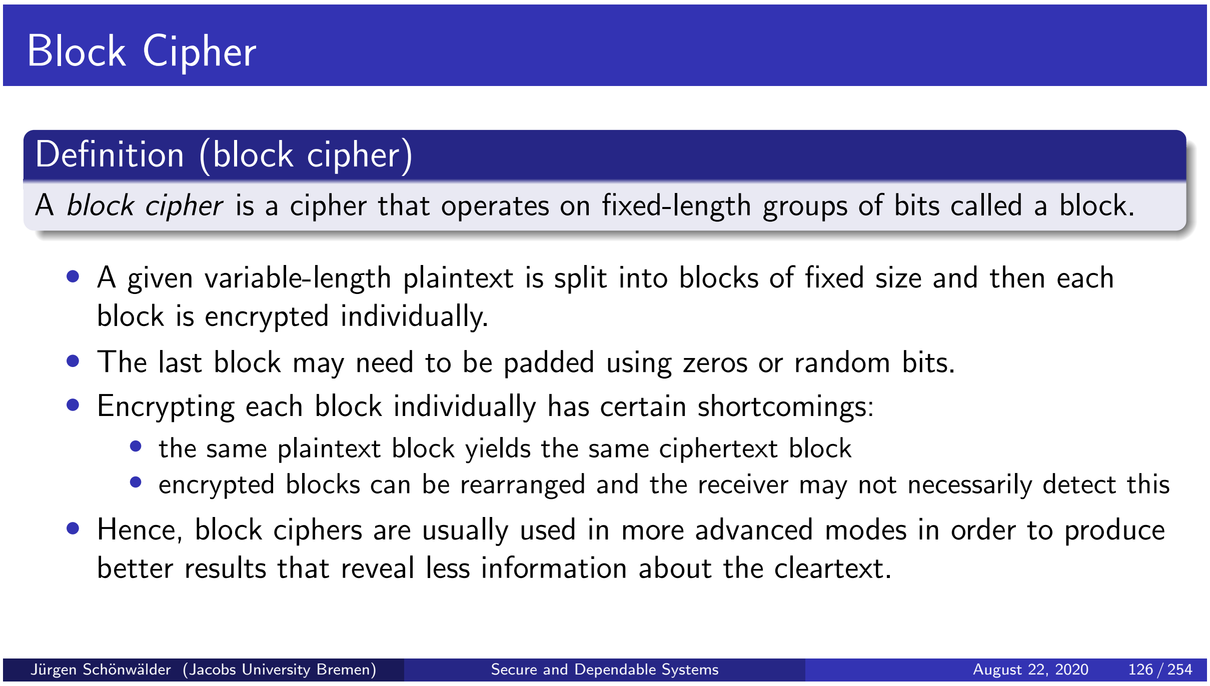


Security of Ciphers

- 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 $\mathrm{E}(m_0)$ or $\mathrm{E}(m_1)$.
- 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 $\frac{1}{2}$.

In practice, the security of a cipher often depends on the properties of the keys. It is common to talk about the length of a key, counted in bits. The length of keys, however, are not easily comparable over different algorithms.

YouTube: 128 Bit or 256 Bit Encryption?



Block Cipher

Definition (block cipher)

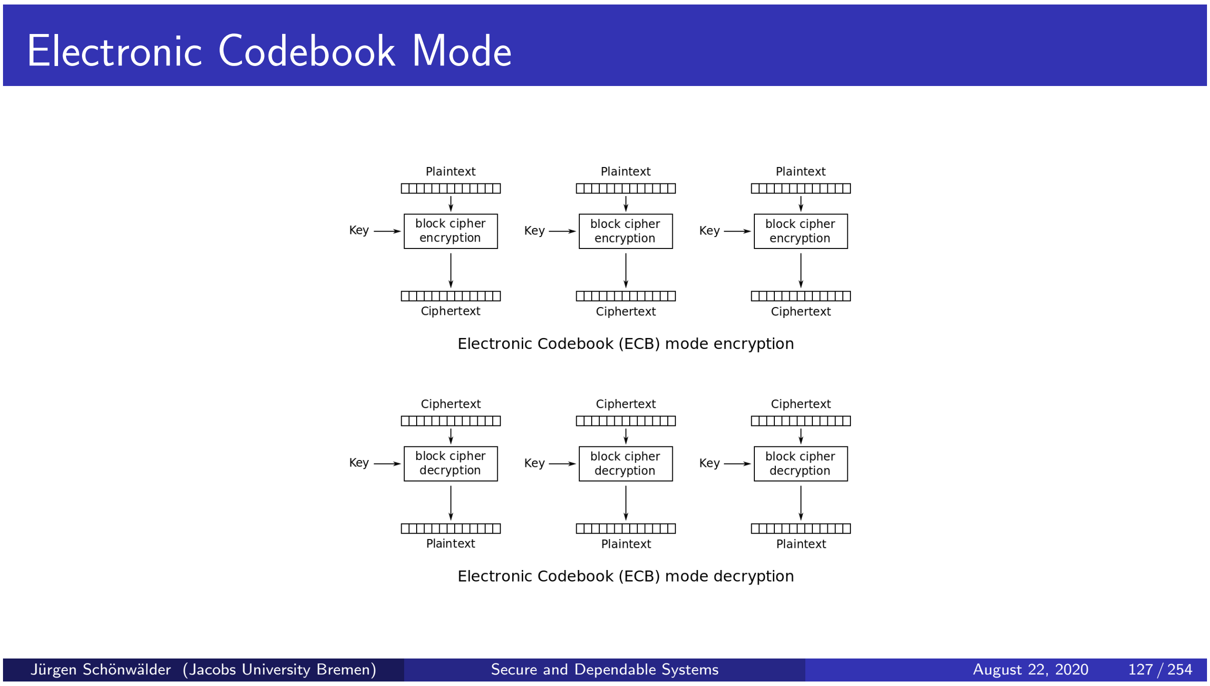
A *block cipher* is a cipher that operates on fixed-length groups of bits called a block.

- 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.

Block ciphers are very widely deployed and used. Before we take a look at concrete ciphers, we first look at the way block ciphers can be applied to cleartexts that span several blocks.

For further information:

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



Electronic Codebook Mode

Electronic Codebook (ECB) mode encryption

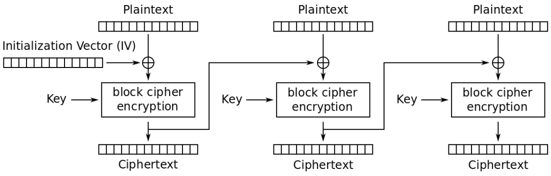
Electronic Codebook (ECB) mode decryption

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 127 / 254

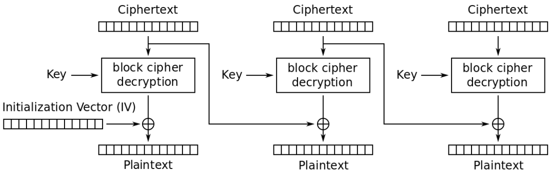
Electronic codebook mode simply slices the input into a sequence of blocks that are all encrypted in isolation.

- 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

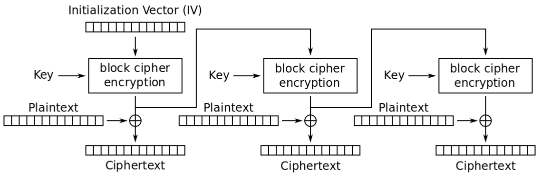
The cipher block chaining mode feeds the ciphertext of block b_i into the encryption of the subsequent block b_{i+1}.

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

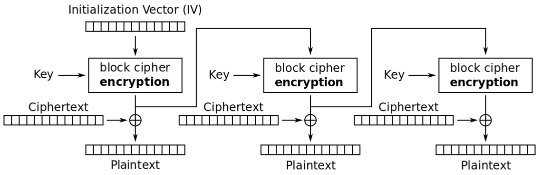
The initialization vector does not have to be secret but it needs to be random and it is ideally only used once. A random number only used once is called a nonce.

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



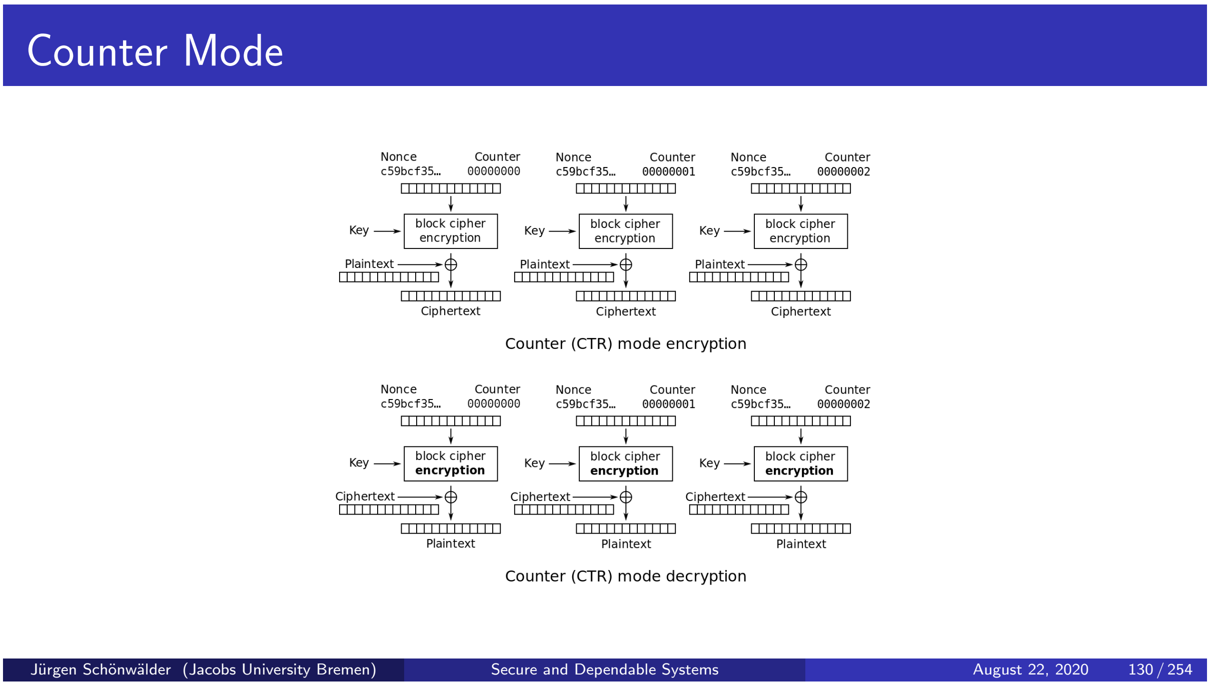
Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

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.

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



Counter Mode

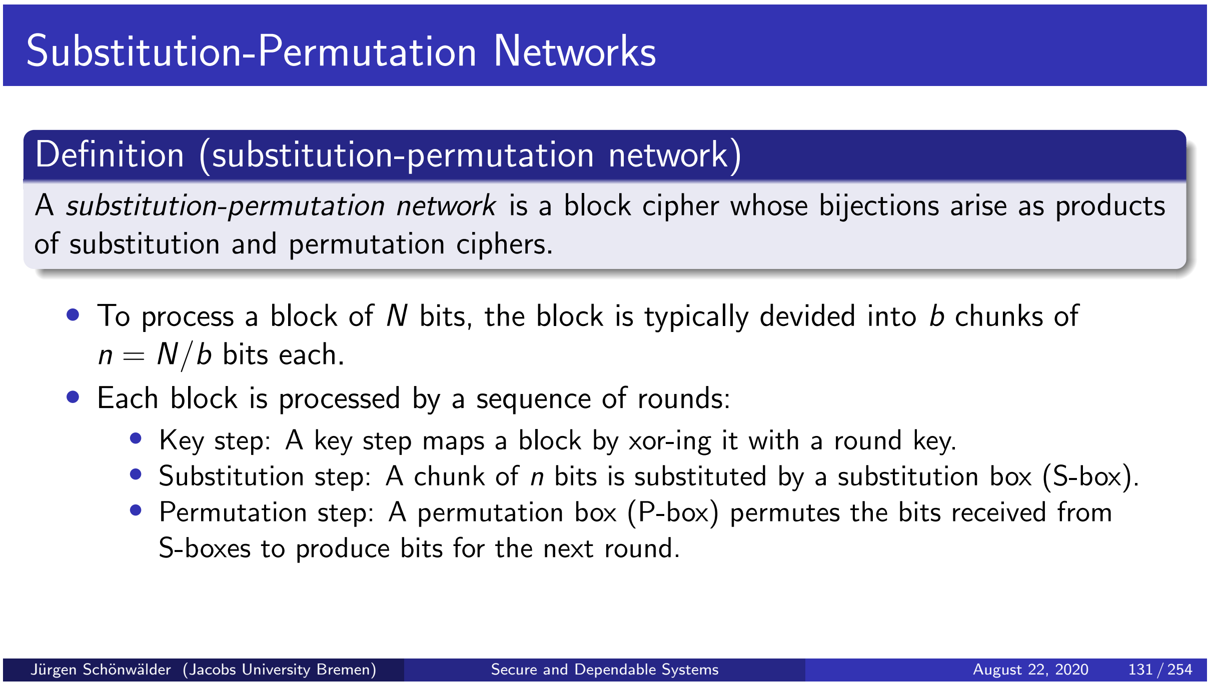
Counter (CTR) mode encryption

Counter (CTR) mode decryption

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 130 / 254

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

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



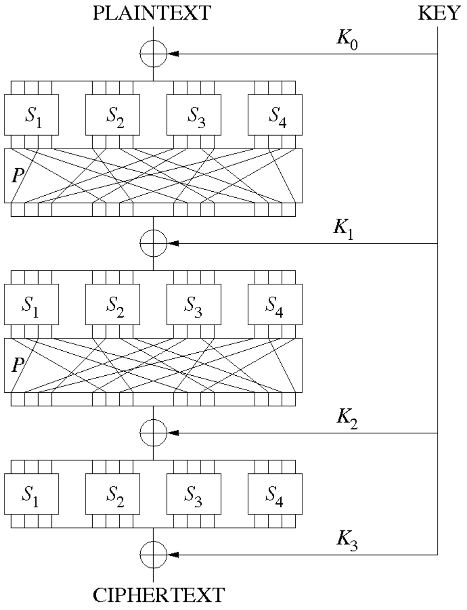
Substitution-Permutation Networks

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 rounds:
  - Key step: A key step maps a block by xor-ing it with a round key.
  - Substitution step: A chunk of n bits is substituted by 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.

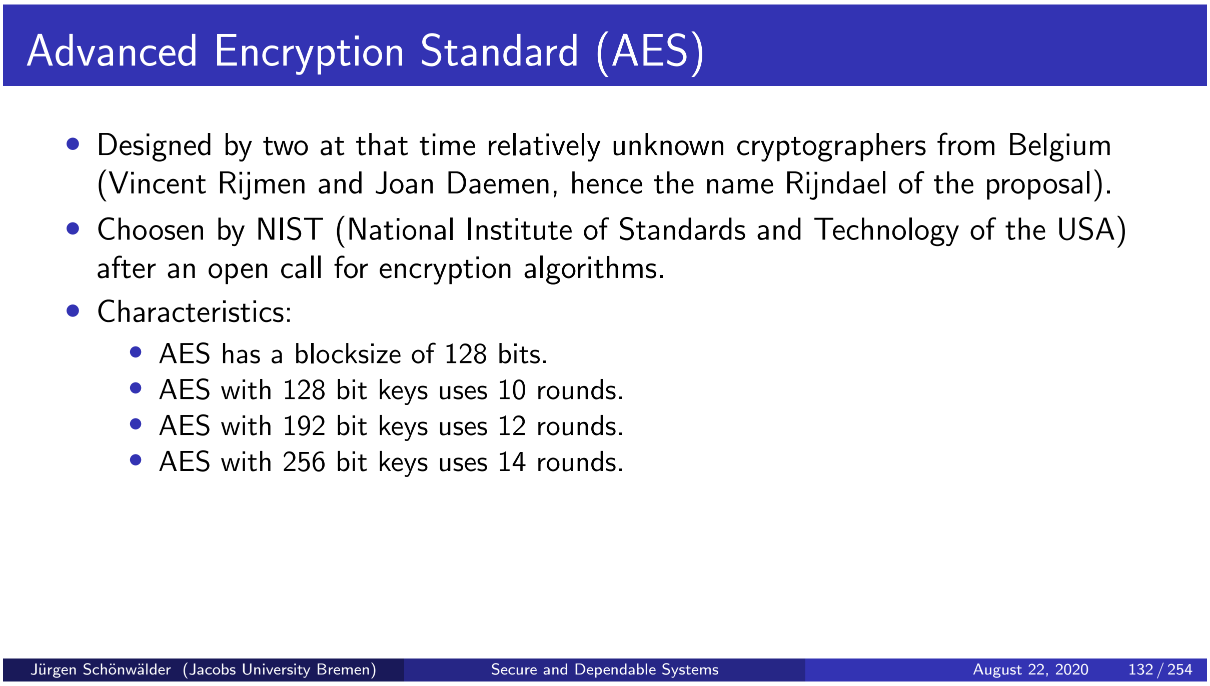
A sketch of a substitution–permutation network with three rounds, encrypting a plaintext block of 16 bits into a ciphertext block of 16 bits. Each S-box processes 4 bits of input while the P-box permutes all 16 bits.



The goal of substitution–permutation networks is to achieve good diffusion and confusion. Diffusion means that changing a single bit of the cleartext should change (statistically) half of the bits in the ciphertext. In other words, even small changes of the cleartext lead to drastic changes of the ciphertext. Confusion means that every bit of the ciphertext should depend on several bits of the key. This obscures the connections between the two. In the last round, the permutation step is often replaced by another key step.

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:
  - AES has a blocksize of 128 bits.
  - AES with 128 bit keys uses 10 rounds.
  - AES with 192 bit keys uses 12 rounds.
  - AES with 256 bit keys uses 14 rounds.

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 [19]. 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.

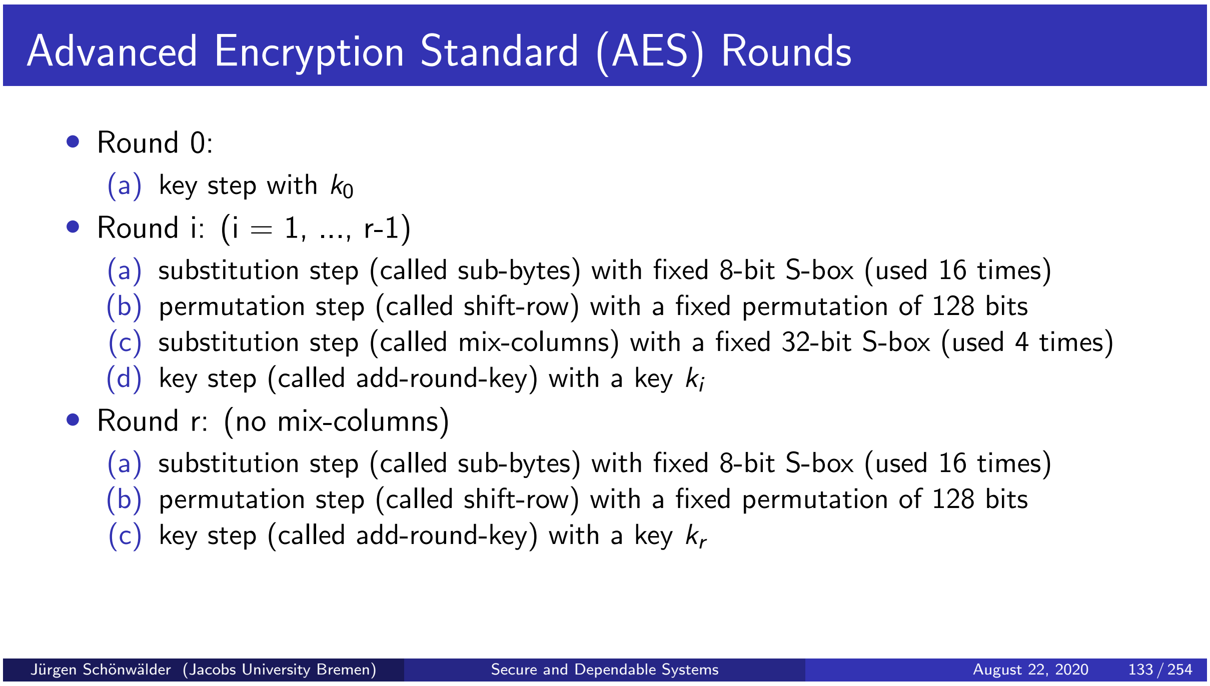
The following `openssl` shell command encrypts the content of the file `welcome.txt` into the file named `welcome.aes`.

```
$ openssl aes-256-cbc -pbkdf2 -in welcome.txt -out welcome.aes
```

As the command indicates, the encryption uses AES with a 256 bit key in cipher block chaining (cbc) mode. Similarly,

```
$ openssl aes-128-ecb -pbkdf2 -in welcome.txt -out welcome.aes
```

will encode the file's content using AES with a 128 bit key in electronic codebook (ecb) mode. Note that the `openssl` command will add random salt, hence the output will differ for each invocation of the command.



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.

The AES algorithm is so widely used that computer hardware often provides hardware support for AES. On Intel processors, the AES-NI (Advanced Encryption Standard New Instructions) provides hardware support for implementing AES rounds. On Linux systems, you can take a look at `/proc/cpuinfo` to check whether your CPUs support `aes`.

YouTube: AES Explained (Advanced Encryption Standard)

For further information:

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

Asymmetric Encryption Algorithms

15 Cryptography Primer

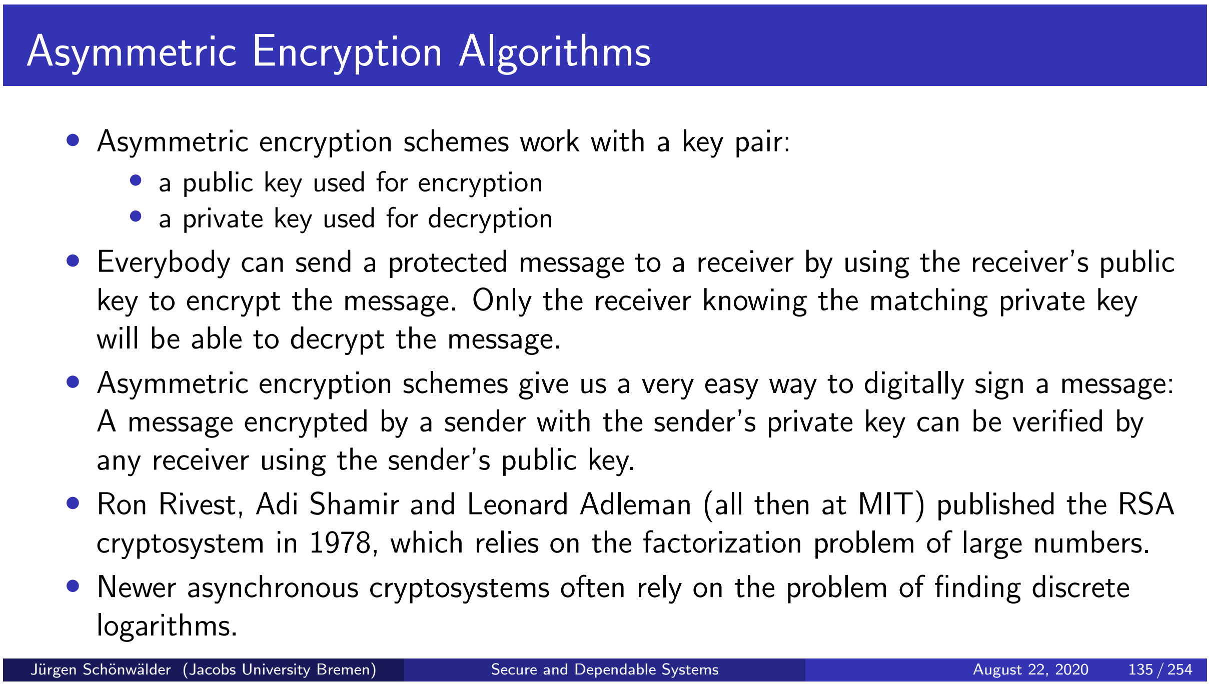
16 Symmetric Encryption Algorithms and Block Ciphers

17 Asymmetric Encryption Algorithms

18 Cryptographic Hash Functions

19 Digital Signatures and Certificates

20 Key Exchange Schemes



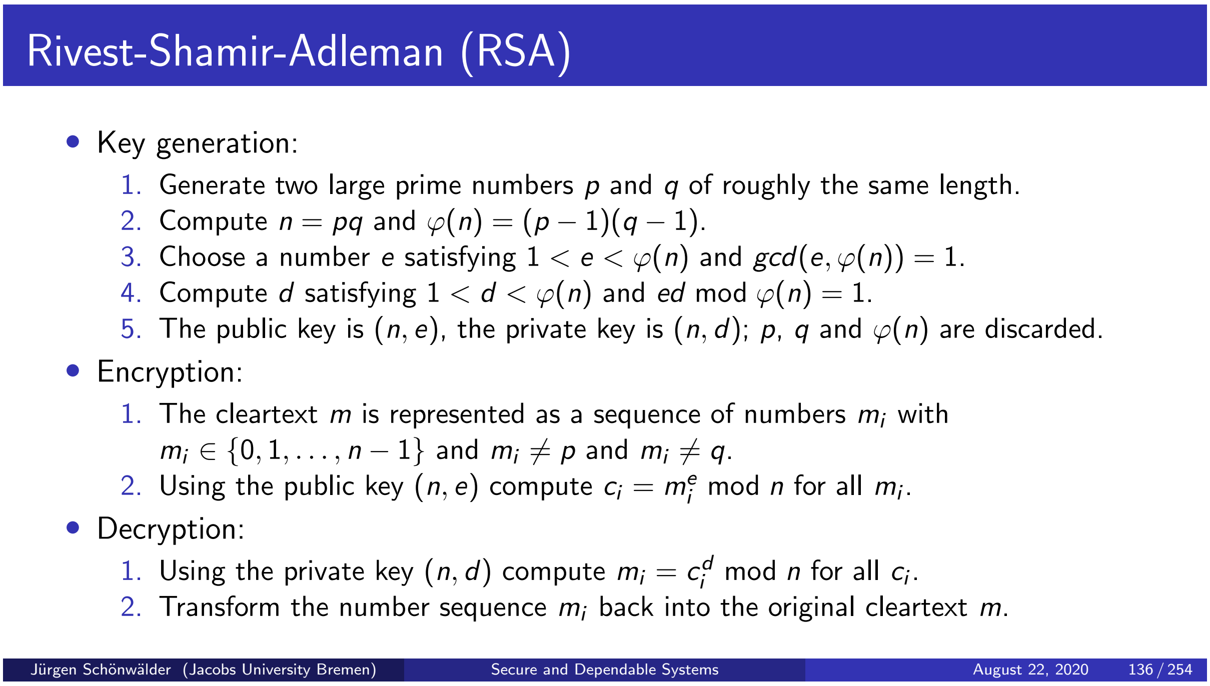
Asymmetric Encryption Algorithms

- 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.

The RSA algorithm was published in 1978 [42] and continues to be used in today's web browsers. Despite this success, the retirement of RSA is visible on the horizon. The reasons is the progress on making quantum computing work; quantum computers will be able to break RSA and hence there is a need to find quantum-resistant cryptographic algorithms.



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 $m_i \in \{0, 1, \ldots, n-1\}$ and $m_i \neq p$ and $m_i \neq q$.
  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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 136 / 254

- 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 `RSA` is converted into the cleartext numbers $m_i = [82, 83, 65]$.
  2. Using the encryption key $(3337, 79)$, we compute

$$\begin{aligned} c_0 &= 82^{79} \bmod 3337 = 274 \\ c_1 &= 83^{79} \bmod 3337 = 2251 \\ c_2 &= 65^{79} \bmod 3337 = 541 \end{aligned}$$

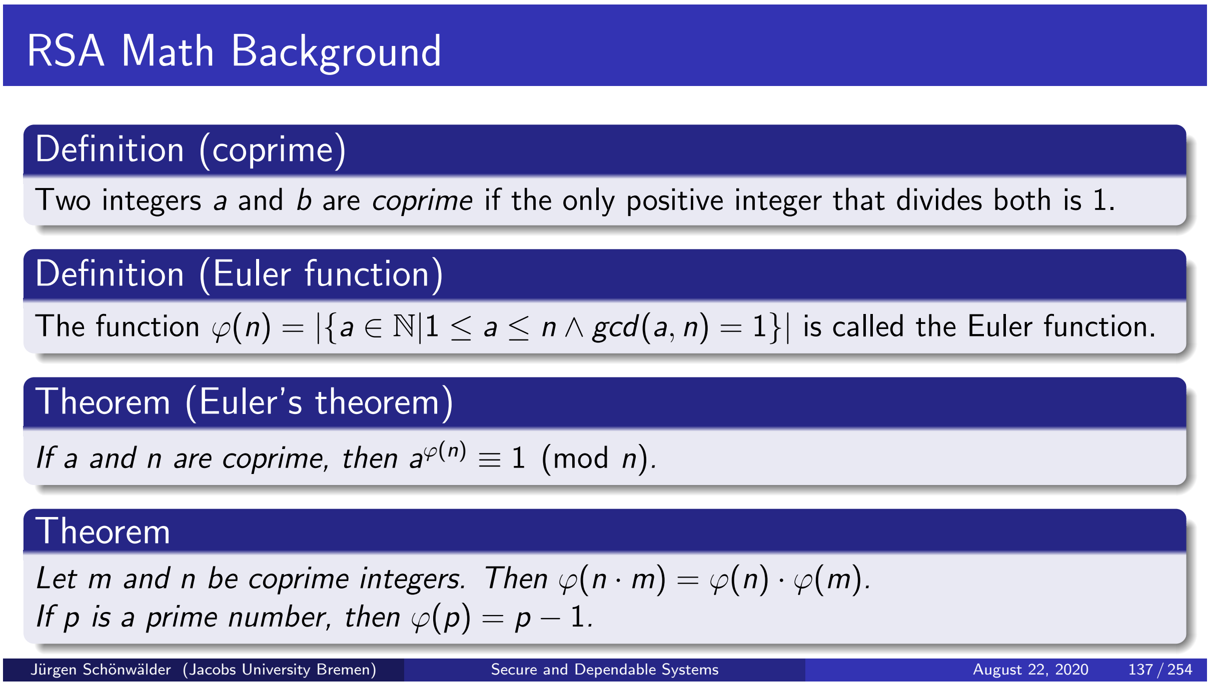
     and we obtain the ciphertext numbers $c_i = [274, 2251, 541]$.
- Decryption:
  1. Using the decryption key $(3337, 1019)$, we compute

$$\begin{aligned} m_0 &= 274^{1019} \bmod 3337 = 82 \\ m_1 &= 2251^{1019} \bmod 3337 = 83 \\ m_2 &= 541^{1019} \bmod 3337 = 65 \end{aligned}$$

     and we obtain the cleartext numbers $m_i = [82, 83, 65]$.
  2. Converting the cleartext numbers back into a string, we get back `RSA`.

YouTube: The RSA Encryption Algorithm (1 of 2: Computing an Example)

YouTube: The RSA Encryption Algorithm (2 of 2: Generating the Keys)



RSA Math Background

Definition (coprime)

Two integers a and b are *coprime* if the only positive integer that divides both is 1.

Definition (Euler function)

The function $\varphi(n) = |\{a \in \mathbb{N} | 1 \leq a \leq n \wedge gcd(a, n) = 1\}|$ is called the Euler function.

Theorem (Euler's theorem)

If a and n are coprime, then $a^{\varphi(n)} \equiv 1 \pmod{n}$.

Theorem

Let m and n be coprime integers. Then $\varphi(n \cdot m) = \varphi(n) \cdot \varphi(m)$.
If p is a prime number, then $\varphi(p) = p - 1$.

We start with Euler's theorem:

$$\begin{aligned} a^{\varphi(n)} &\equiv 1 \pmod{n} \\ a^{k \cdot \varphi(n)} &\equiv 1 \pmod{n} \\ a^{k \cdot \varphi(n)+1} &\equiv a \pmod{n} \end{aligned}$$

We want two keys e and d such that $a^{ed} \equiv a \mod n$. This means we want $ed = k\varphi(n) + 1$.

Lets restrict n to $n = pq$ for two prime numbers p and q. Obviously, p and q are coprime and hence applying our theorems we get $\varphi(n) = \varphi(pq) = \varphi(p)\varphi(q) = (p - 1)(q - 1)$. With that, we obtain:

$$ed = k\varphi(n) + 1$$

If we take this modulo $\varphi(n)$, we get:

$$ed \equiv 1 \pmod{\varphi(n)}$$

Hence, we can choose e and then find a d such that $ed \equiv 1 \mod \varphi(n)$.

For finding d, we can use the extended Euclidian algorithm, which solves the following problem: Given two integers a and b, calculate the d and s and t such $d = gcd(a, b)$ that $d = s \cdot a + t \cdot b$.

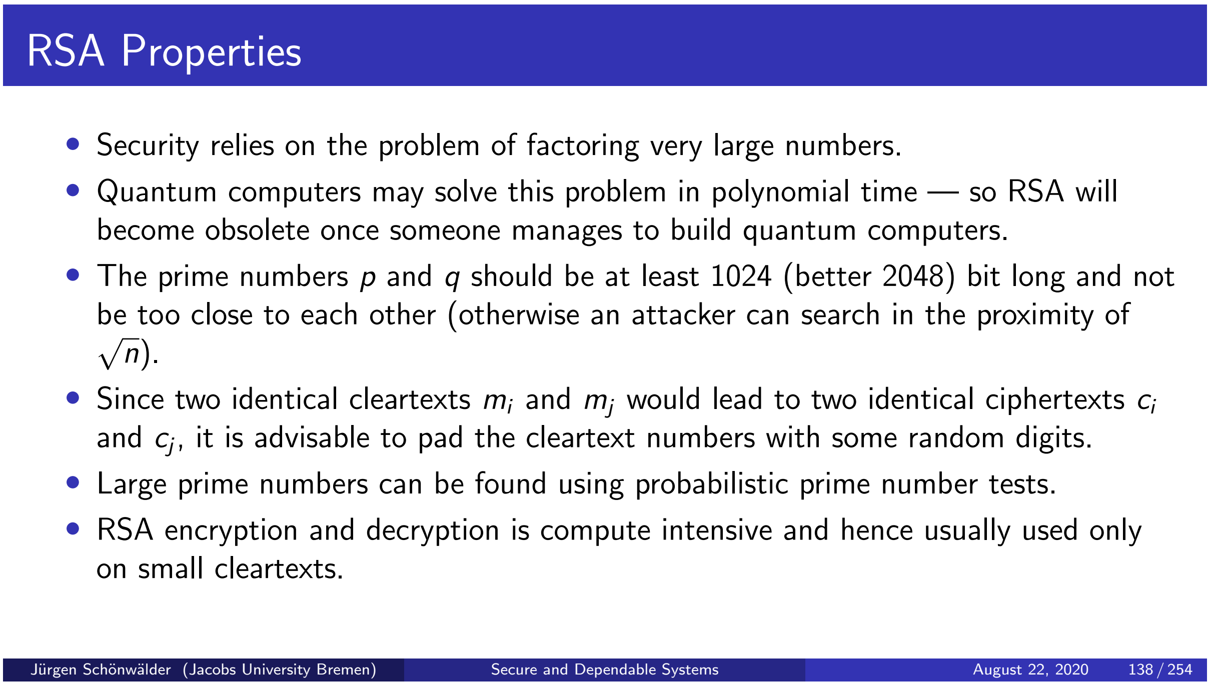
If we call the extended Euclidian algorithm with $a = e$ and $b = \varphi(n)$, we get

$$1 = s \cdot e + t \cdot \varphi(n)$$

and if we take this modulo $\varphi(n)$, the term with $\varphi(n)$ disappears and we get:

$$1 \equiv s \cdot e \pmod{\varphi(n)}$$

YouTube: RSA – The Math



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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 138 / 254

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.

Here is an implementation of the extended Euclidian greatest common divisor (gcd) algorithm in Haskell:

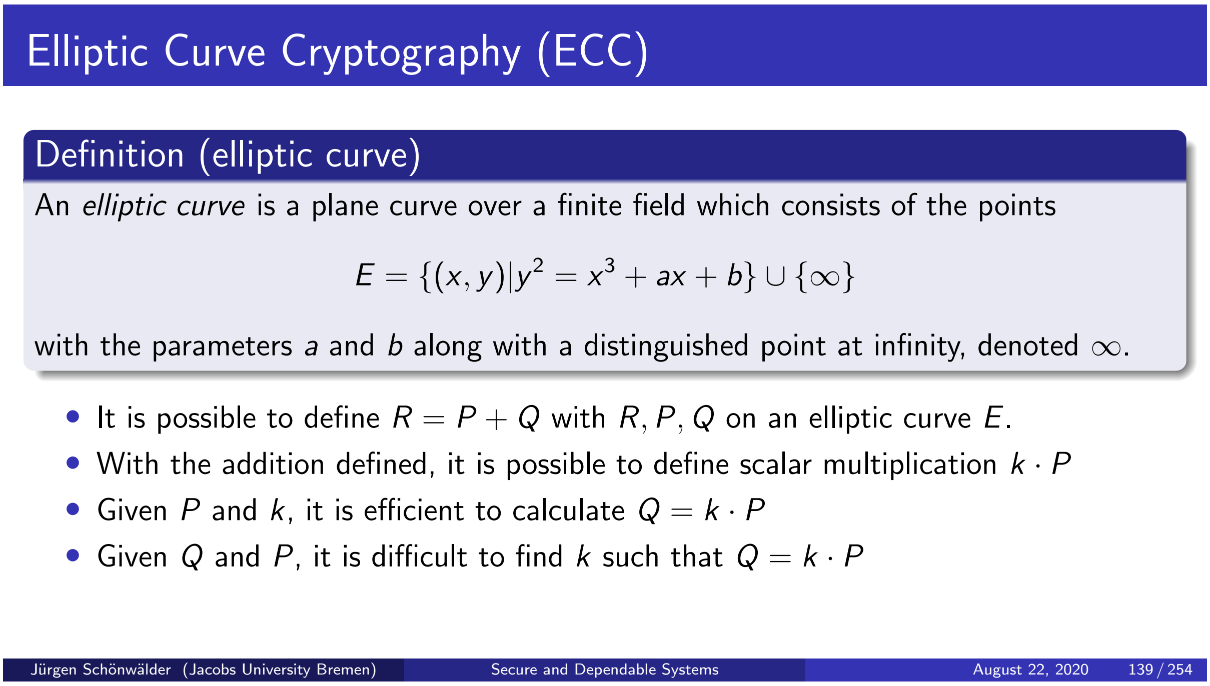
```haskell
-- for a b calculate d and s and t such that d = gcd(a,b) and d = s*a + t*b
egcd :: Integer -> Integer -> (Integer, Integer, Integer)
egcd a 0 = (abs a, signum a, 0)
egcd a b = (d, t, s - (a `div` b) * t)
    where (d, s, t) = egcd b (a `mod` b)
```

And here is an implementation of a fast modular exponentiation algorithm in Haskell:

```haskell
-- calculate (a^b) `mod` m efficiently
modExp :: Integer -> Integer -> Integer -> Integer
modExp _ 0 _ = 1
modExp a b m
  | even b    = (r * r) `mod` m
  | otherwise = (a * r * r) `mod` m
  where r = modExp a (b `div` 2) m
```

RSA key generation and encryption/decryption can be done using `openssl` as follows:

```
  $ openssl genrsa -aes256 -out private.key 8912
  $ openssl rsa -in private.key -pubout -out public.key

  $ echo "Welcome to Secure and Dependable Systems" > welcome.txt
  $ openssl rsautl -encrypt -pubin -inkey public.key -in welcome.txt -out welcome.rsa
  $ openssl rsautl -decrypt -inkey private.key -in welcome.rsa -out plaintext.txt
```



Elliptic Curve Cryptography (ECC)

Definition (elliptic curve)

An *elliptic curve* is a plane curve over a finite field which consists of the points

$$E = \{(x, y) | y^2 = x^3 + ax + b\} \cup \{\infty\}$$

with the parameters a and b along with a distinguished point at infinity, denoted ∞.

- It is possible to define $R = P + Q$ with R, P, Q on an elliptic curve E.
- With the addition defined, it is possible to define scalar multiplication $k \cdot P$
- Given P and k, it is efficient to calculate $Q = k \cdot P$
- Given Q and P, it is difficult to find k such that $Q = k \cdot P$

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 139 / 254

ECC plays an important role today since much shorter ECC keys achieving the same security as much longer RSA keys. There is a large pool of eliptic curves that have been proposed, some are suspected to be bad ones (i.e., the parameters may have hidden backdoors), others are currently believed to be good ones (but may not be anymore when you read this).

RFC 7748 [2] (published in 2016) defines elliptic curves for security applications and it recommends Curve25519 and Curve448. These are Montgomery curves of the form $v^2 = u^3 + A \cdot u^2 + u$.

- Curve25519: $v^2 = u^3 + 486662 \cdot u^2 + u \mod p$ with $p = 2^{255} - 19$
- Curve448: $v^2 = u^3 + 156326 \cdot u^2 + u \mod p$ with $p = 2^{448} - 2^{224} - 1$

For a short introduction into the basics of ECC cryptography, watch the following videos.

YouTube: Elliptic Curve Diffie Hellman

YouTube: Martijn Grooten - Elliptic Curve Cryptography for those who are afraid of maths

Cryptographic Hash Functions

15 Cryptography Primer

16 Symmetric Encryption Algorithms and Block Ciphers

17 Asymmetric Encryption Algorithms

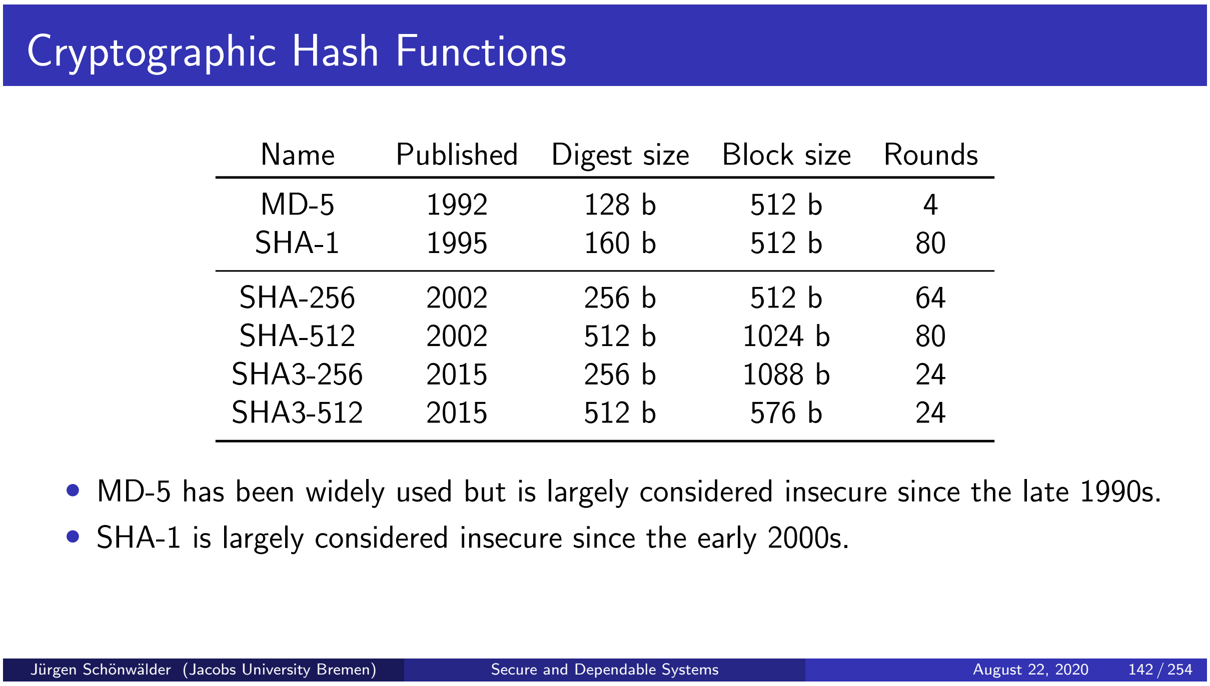
18 Cryptographic Hash Functions

19 Digital Signatures and Certificates

20 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.

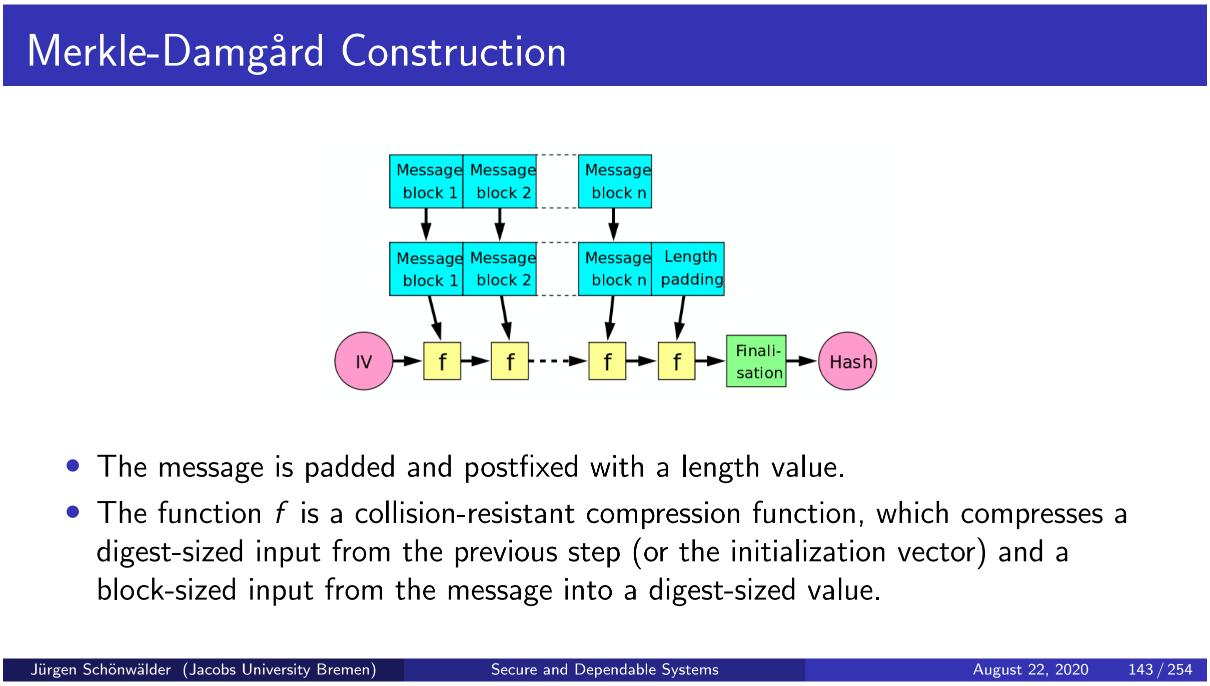
Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 142 / 254

Unix systems often came with command line tools to calculate hash values such as `shasum`. The `openssl` command can also be used to calculate hash values.

```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ shasum -a 256 welcome.txt
b34008c3d75d00108fb669366ebdb407b893ffbebdbd265e741fd62349db9868  welcome.txt
$ openssl sha256 welcome.txt
SHA256(welcome.txt)= b34008c3d75d00108fb669366ebdb407b893ffbebdbd265e741fd62349db9868
```

The `shasum` tool have been written to produce checksums for a list of files and to verify a list of files against previously computed checksums.

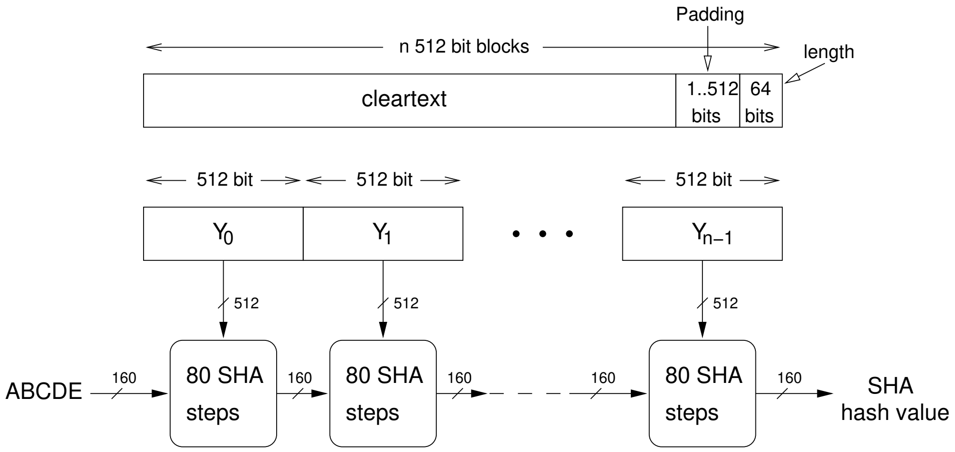
```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ shasum -a 256 welcome.txt > welcome.sha256
$ shasum -c welcome.sha256
welcome.txt: OK
```



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.

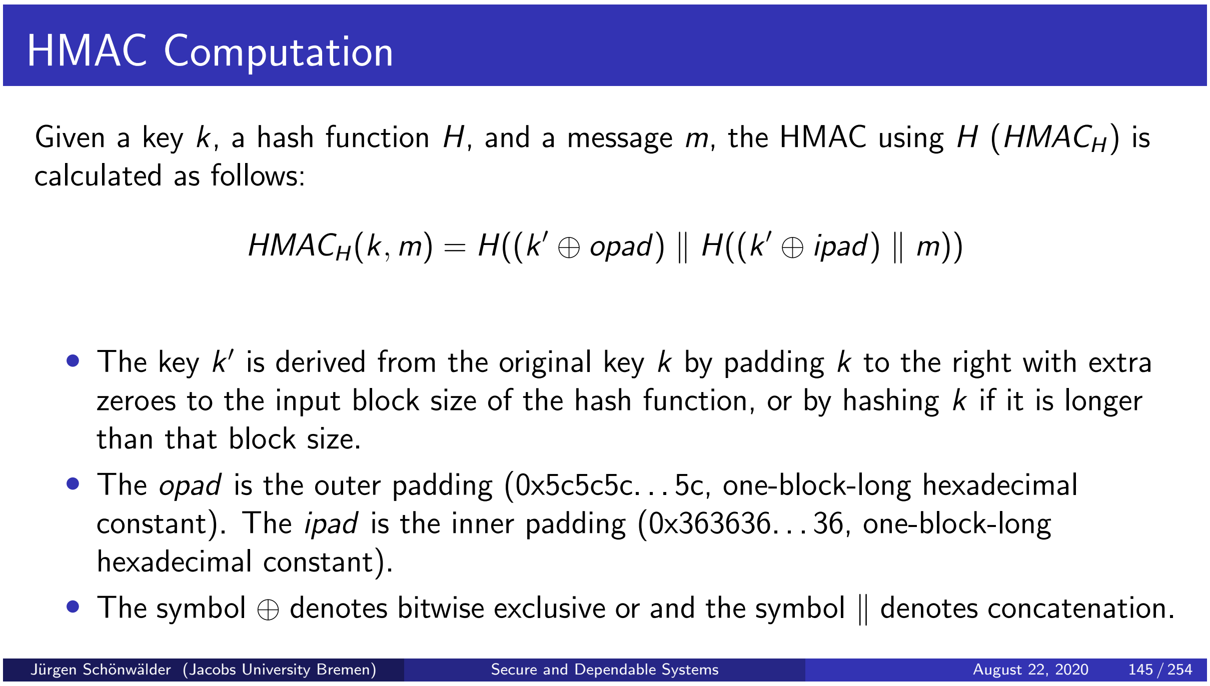
Example (SHA-1):



YouTube: SHA: Secure Hashing Algorithm

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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 145 / 254

HMACs [20] 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.

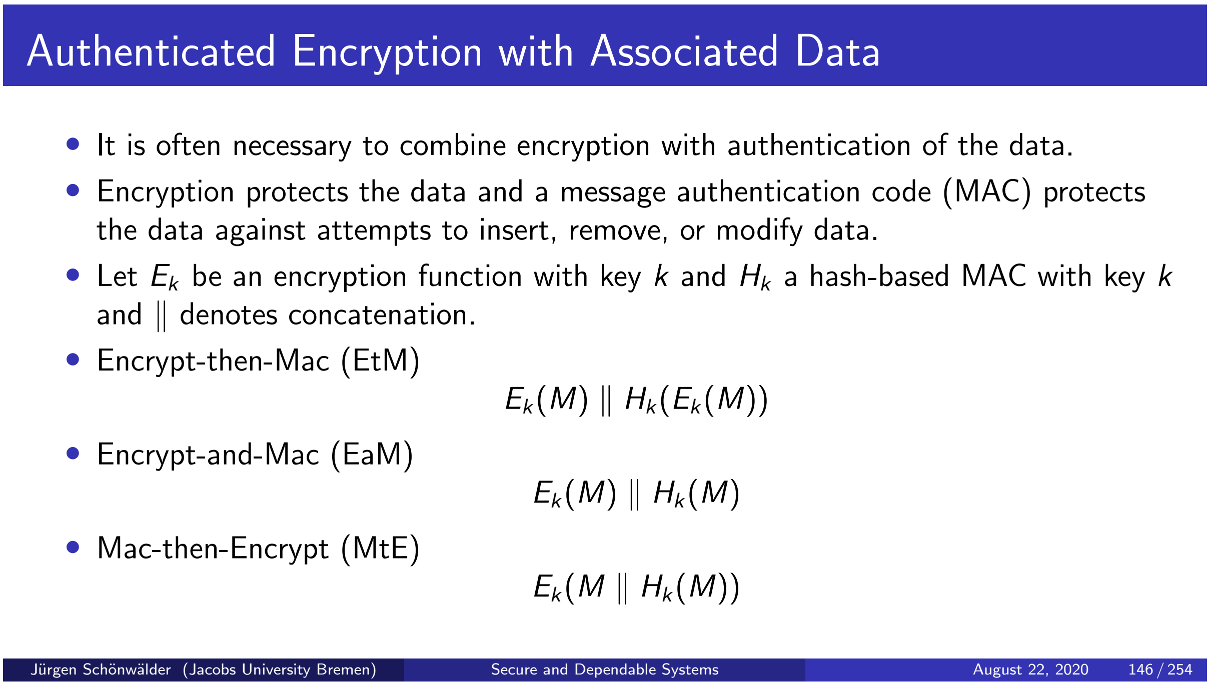
The design of HMAC avoids attacks that are possible on simpler constructions:

- $HMAC_H = H(K||m)$ 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 [40] defines among others the cipher suites `TLS_AES_128_GCM_SHA256` or `TLS_AES_256_GCM_SHA384`. The AEAD algorithms are defined in RFC 5116 [34].

The slide basically talks about authenticated encryption. The associated data is additional plaintext data that is not encrypted but covered by the hash. This is a common situation in communication protocols where the payload carried in messages is encrypted while some additional message fields remain unencrypted in order to organize the forwarding of messages. In addition, it is often required to ensure that messages are “fresh”, i.e., not a replay of some old messages. Hence, the AEAD interface defined in RFC 5116 [34] consists of two functions

$$\begin{aligned} enc &: (K \times N \times P \times A) \rightarrow C \\ dec &: (K \times N \times C \times A) \rightarrow P \end{aligned}$$

where K is is key, N is a nonce (a random distinct unused value), P is some plaintext, A is associated data, and C is ciphertext. Newer security protocols usually use AEADs instead of basic cryptographic algorithms. (The functions enc and dec can also indicate failure situations.)

The different AEAD techniques have different properties [29, 9]. In general, Encrypt-then-Mac is preferred by most cryptographers since it protects against chosen ciphertext attacks and avoids any confidentiality issues arising from the MAC of the cleartext message.

Digital Signatures and Certificates

15 Cryptography Primer

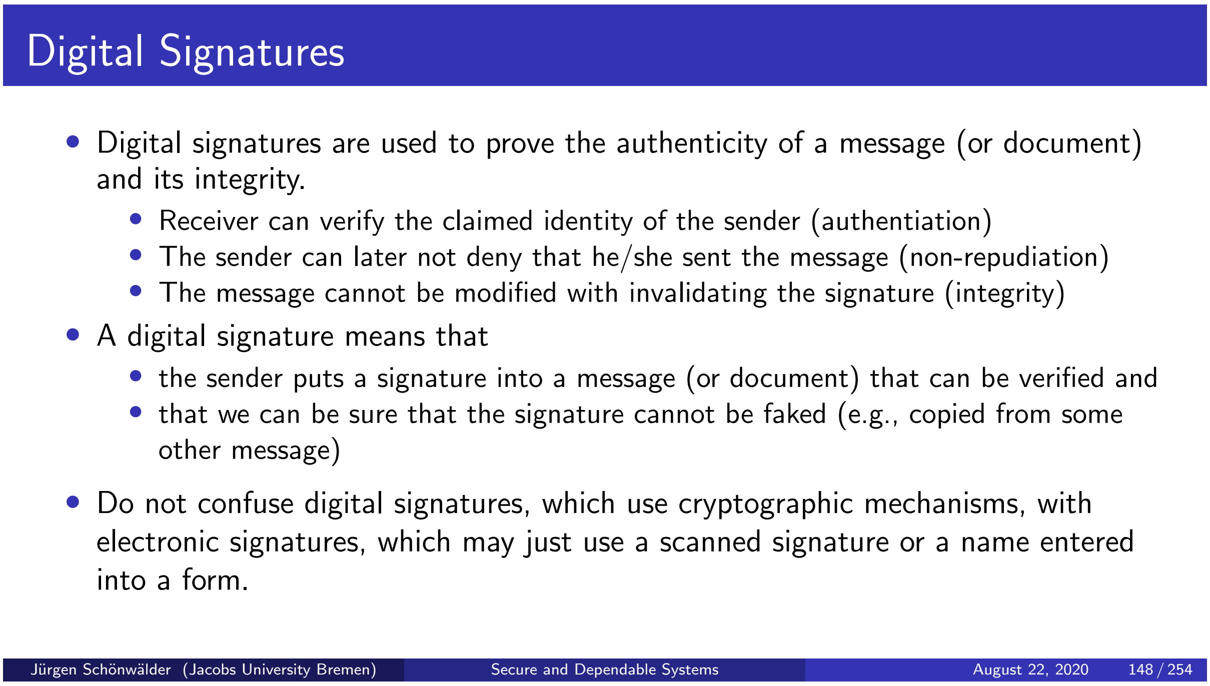
16 Symmetric Encryption Algorithms and Block Ciphers

17 Asymmetric Encryption Algorithms

18 Cryptographic Hash Functions

19 Digital Signatures and Certificates

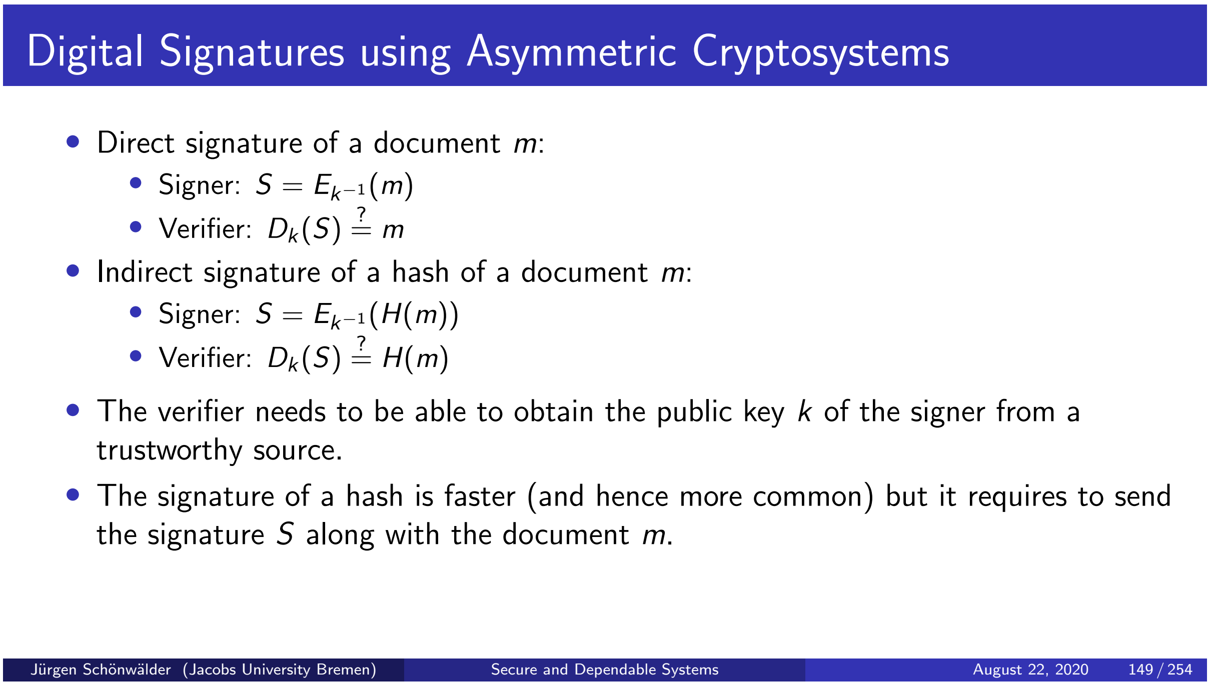
20 Key Exchange Schemes



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.

Students often send me emails asking me to digitally sign some forms. Well, I usually have to decline these requests since digitally signing a form requires that there is an infrastructure in place so that others can obtain the cryptographic keys necessary to check my digital signature. Without such an infrastructure, a digital signature is pretty useless. What student often want is in fact an electronic signature but that is pretty pointless as well since pretty much everybody can learn how to copy a scan of my signature into a document.



Digital Signatures using Asymmetric Cryptosystems

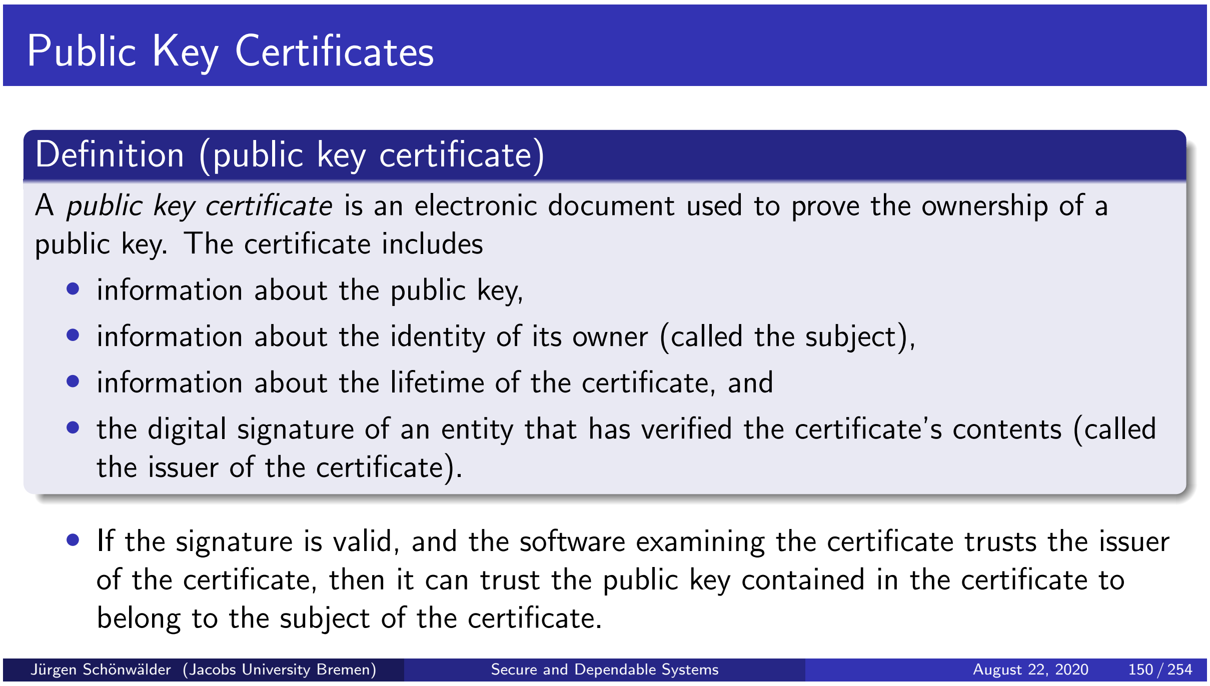
- Direct signature of a document m:
  - Signer: $S = E_{k^{-1}}(m)$
  - Verifier: $D_k(S) \overset{?}{=} m$
- Indirect signature of a hash of a document m:
  - Signer: $S = E_{k^{-1}}(H(m))$
  - Verifier: $D_k(S) \overset{?}{=} H(m)$
- The verifier needs to be able to obtain the public key k of the signer 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.

In practice, digital signatures most often work with hashes of documents, i.e., they are indirect signatures. Instead of signing a potential long electronic document, a cryptographic hash is calculated and then signed with the singer's private key. The received of the document and the signature can verify the signature by obtaining the signer's public key, calculating the hash of the document, and comparing the decrypted signature with the locally calculated hash.

Digital signatures using asymmetric cryptosystems are in general simple but the catch is that the verifier needs to obtain *and trust* the signer's public key. If Mallory creates a key pair and she manages to make Bob believe the public part of the key pair belongs to Alice, then she can send messages under the identity of Alice and Bob will believe them to be authentic. Another problem arises if private keys are leaked or broken (or expired). Such an event can effectively turns all past signatures useless. So Bob not only needs to trust that Alice's key is in fact Alice's key, he also need to verify at the time he uses the key that the key is still valid and has not been revoked yet. This all turns something that is conceptually simple into something that is astonishingly complex.

It is possible to create digital signatures of arbitrary files using the following `openssl` commands:

```
# create an rsa keypair and extract the public key
openssl genrsa -aes256 -out private.key 8912
openssl rsa -in private.key -pubout -out public.key

# create a digital signature of a given file
file=sads-notes.pdf
openssl dgst -sha256 -sign private.key -out signature.sha256 $file
# optionally convert the signature to base64
openssl enc -base64 -in sign.sha256 -out signature.sha256.base64

# verify the digital signature of a given file
file=sads-notes.pdf
openssl dgst -sha256 -verify public.key -signature sign.sha256 $file
```



Public Key Certificates

Definition (public key certificate)

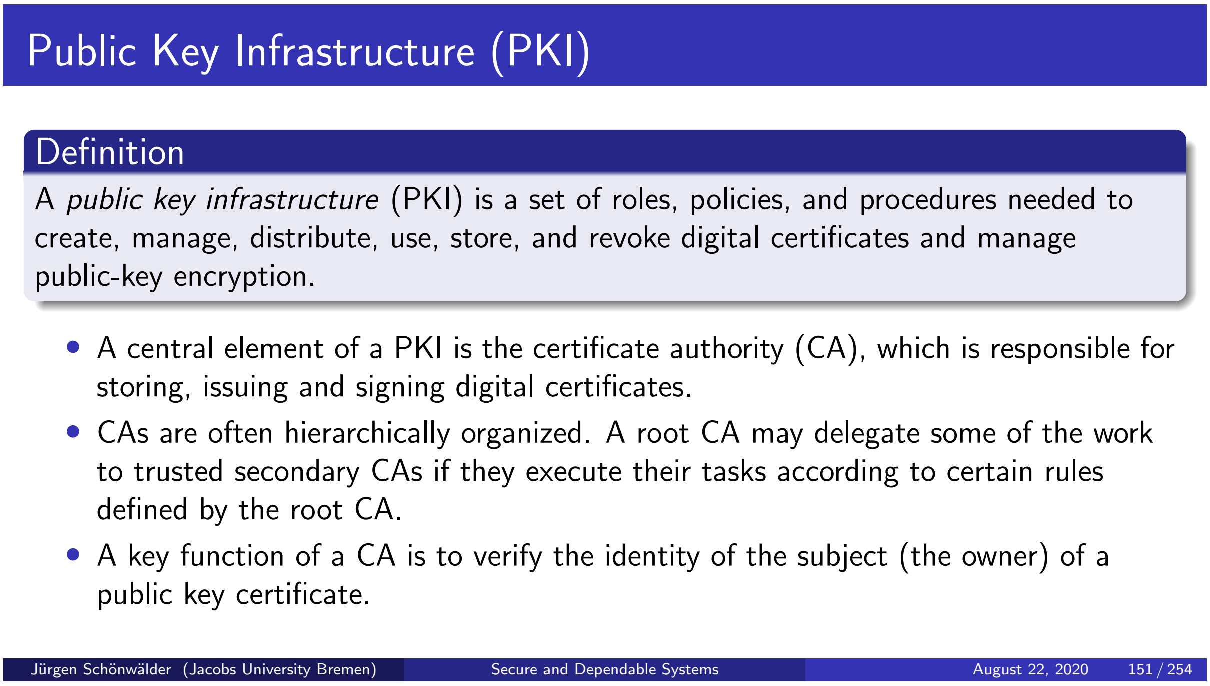
A *public key certificate* is an electronic document used to prove the ownership of a public key. The certificate includes

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

- 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.

Obviously, to trust a given certificate, you need to trust the issuer of that certificate. This may require to trust the issuer of the issuer of the certificate and so on. This results in a chain of trust relationships that must be rooted somewhere.

Some people believe into hierarchical trust models, where trust is following existing hierarchical structures in enterprises or governments. Other people believe into decentralized and self-organizing trust networks, where trust is obtained on a peer-to-peer basis and for some peers trust may be transitive ("I am willing to trust the friends of my best friend."). Both models seem to have their place in different contexts.



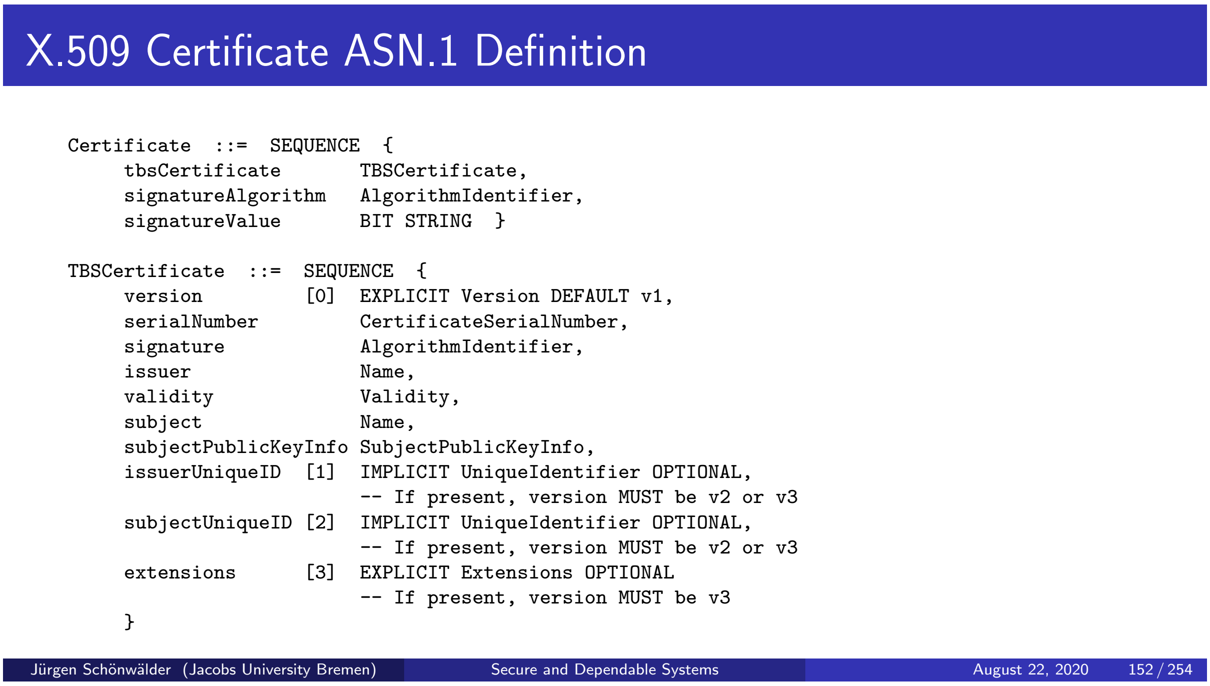
Public Key Infrastructure (PKI)

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.

At Jacobs University, the IT department can sign certificates. They obtained the right to sign certificates from the national research network DFN. For every signature, they need to verify who requested it. Even though I know some of them for many years, I regularly go there to show my passport and to prove my identity. The reason is that there are well-defined procedures and must be followed (and Germans tend to take such procedures very serious).



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
     }
```

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 152 / 254

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

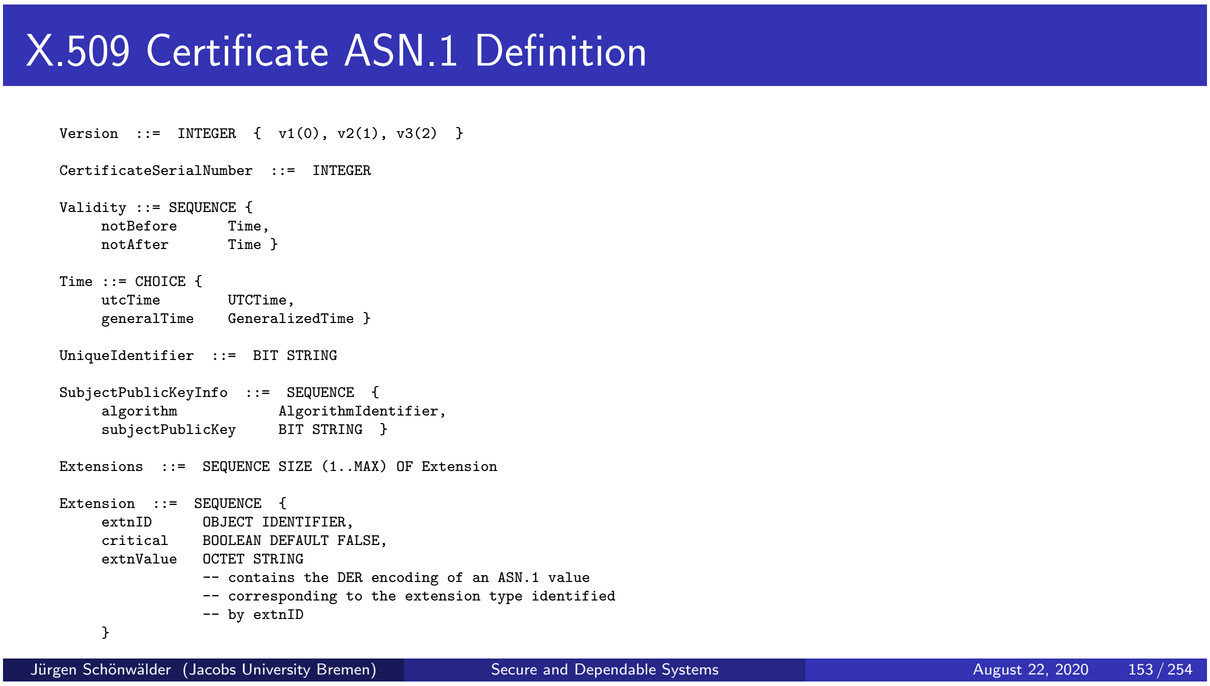
```
Certificate  ::=  SEQUENCE  {
   tbsCertificate       TBSCertificate,
   signatureAlgorithm   AlgorithmIdentifier,
   signatureValue       BIT STRING  }
```

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
     }
```

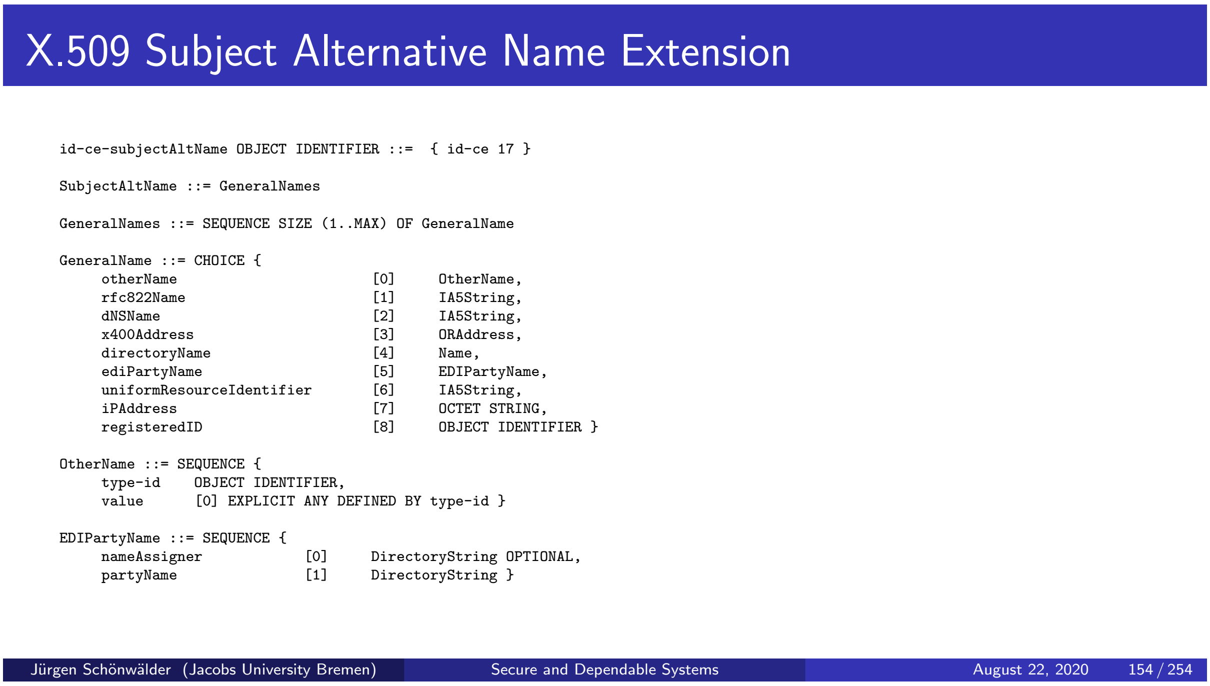
Implementations often store certificates in .crt files. Using `openssl`, it is possible to download certificates from arbitrary web sites. A .crt file can be converted into human readable text using an `openssl` shell commands as well:

```
$ HOST='cnds.jacobs-university.de'
$ echo | openssl s_client -servername $HOST -connect $HOST:443 \
    | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > /tmp/certificate.crt
$ openssl x509 -in /tmp/certificate.crt -text -noout
```

There are a number of online tools that do details analysis of certificates used by web sites. A classic online tool is provided by Qualys, Inc.: https://www.ssllabs.com/ssltest/.

The following page shows a certificate obtained and converted to human readable text using the `openssl` commands.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            1f:c1:cd:3f:42:51:99:82:07:7f:e1:3d
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Global Issuing CA
        Validity
            Not Before: Sep 19 13:31:30 2018 GMT
            Not After : Dec 21 13:31:30 2020 GMT
        Subject: C = DE, ST = Bremen, L = Bremen, O = Jacobs University Bremen gGmbH, CN = cnds.jacobs-university.de
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:9b:03:c1:e2:84:ca:03:44:a6:a9:cc:21:d9:f7:
                    88:20:1a:3f:1b:9f:7e:90:2a:2a:ed:63:6b:e6:cc:
                    f3:71:1b:44:cb:51:bd:01:dc:68:c1:86:3f:5a:59:
                    4f:20:12:74:57:bd:6d:c6:39:19:fb:ea:f5:05:12:
                    0f:94:6e:77:8a:5a:7d:14:be:53:b0:9c:dd:11:3e:
                    b8:5c:9e:d1:94:57:ee:c5:d0:92:86:53:9e:e5:cf:
                    a3:d3:8e:6a:01:65:d3:21:bc:b3:f1:78:b9:ff:59:
                    7e:e2:8f:86:08:01:c6:ca:dc:2a:0b:ff:4e:5a:3a:
                    cd:fb:c9:b7:e8:1c:c0:9c:6f:33:7e:95:1c:99:f7:
                    03:4c:55:2c:86:5b:54:81:28:30:e6:d0:cd:e2:8c:
                    99:3e:23:22:aa:5f:de:53:84:d9:29:8b:c6:ed:ae:
                    73:90:87:4c:00:5c:cf:43:ca:3b:e3:fe:5b:ba:ef:
                    e8:b7:9d:97:81:8b:3a:05:01:fc:da:29:06:31:22:
                    b4:ca:de:ce:72:ee:cf:70:d7:a7:90:91:bc:94:4a:
                    12:ed:91:bc:d9:45:47:3f:14:7e:7b:00:69:49:a3:
                    8b:28:cd:f2:2e:be:82:39:01:c3:dc:05:d5:96:95:
                    5b:5c:96:eb:53:9f:d3:23:ee:41:b4:11:34:27:87:
                    5a:1b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Certificate Policies:
                Policy: 2.23.140.1.2.2
                Policy: 1.3.6.1.4.1.22177.300.30
                Policy: 1.3.6.1.4.1.22177.300.1.1.4
                Policy: 1.3.6.1.4.1.22177.300.1.1.4.3.8
                Policy: 1.3.6.1.4.1.22177.300.2.1.4.3.8

            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:
                36:E8:F1:C2:B6:60:12:85:1A:46:BC:AF:50:02:99:D6:13:E6:3E:00
            X509v3 Authority Key Identifier:
                keyid:6B:3A:98:8B:F9:F2:53:89:DA:E0:AD:B2:32:1E:09:1F:E8:AA:3B:74

            X509v3 Subject Alternative Name:
                DNS:cnds.jacobs-university.de
            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://cdp1.pca.dfn.de/dfn-ca-global-g2/pub/crl/cacrl.crl

                Full Name:
                  URI:http://cdp2.pca.dfn.de/dfn-ca-global-g2/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/dfn-ca-global-g2/pub/cacert/cacert.crt
                CA Issuers - URI:http://cdp2.pca.dfn.de/dfn-ca-global-g2/pub/cacert/cacert.crt

    Signature Algorithm: sha256WithRSAEncryption
         84:41:ee:16:69:d0:df:72:f0:2d:7a:a3:b4:6f:98:6f:d4:cf:
         06:0e:26:00:26:14:52:62:7f:9e:c1:47:a7:8b:c3:62:3a:9e:
         22:ee:2c:2f:56:68:b5:2d:3b:8e:f7:be:fc:06:85:3c:1e:3c:
         37:67:3f:a1:48:37:c5:17:9c:5c:96:ab:33:55:ec:a2:94:78:
         76:d9:d3:b1:d0:d8:a0:eb:22:65:93:a8:aa:22:e1:c6:12:62:
         0f:b1:72:4b:67:66:95:c2:19:88:cc:68:71:56:e3:23:f6:89:
         26:ad:cb:18:1f:11:52:69:a3:c1:95:9c:c1:14:2d:ad:01:38:
         17:23:7a:38:bc:6d:c0:8f:0a:18:ac:82:bc:a6:c6:fe:13:7c:
         25:f5:3f:92:11:a7:2e:fe:ae:79:45:62:fa:39:0d:e7:45:04:
         d7:c2:2a:9b:c1:b8:1c:a1:93:bd:d7:30:3f:7a:34:93:f9:44:
         1b:29:1a:38:97:56:4a:98:48:82:cb:71:26:ca:1b:86:f0:36:
         23:12:5b:fb:ba:6c:63:9f:3f:1f:46:b8:ea:ae:62:67:ce:89:
         20:86:62:09:8e:8b:53:1e:0d:12:50:5b:5f:f4:1c:b3:9f:af:
         7b:a5:75:78:ca:bf:ae:da:81:28:4d:a9:64:73:38:fd:de:cb:
         46:53:a9:a6
```

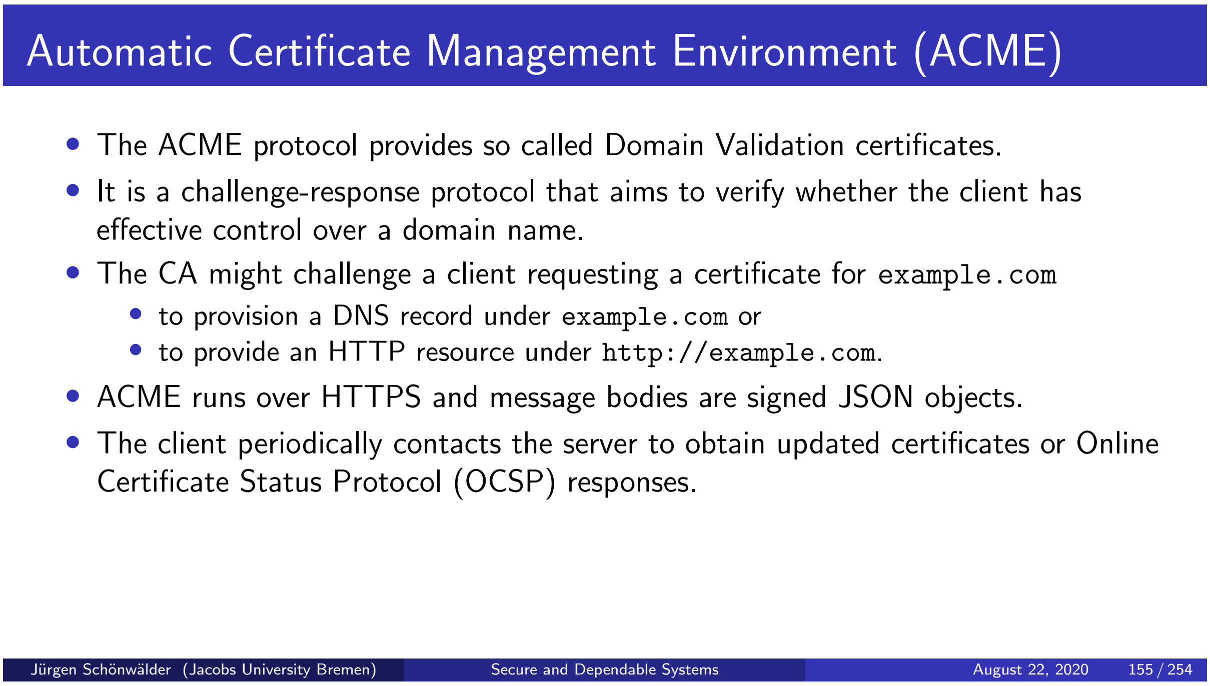


X.509 Subject Alternative Name Extension

```
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 an 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 `cnds.jacobs-university.de`. For further details about the X.509 format, see RFC 5280 [14].

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 [43].



Automatic Certificate Management Environment (ACME)

- The ACME protocol provides so called Domain Validation certificates.
- It is a challenge-response protocol that aims to verify whether the client has effective control over a domain name.
- The CA might challenge a client requesting a certificate for `example.com`
  - to provision a DNS record under `example.com` or
  - to provide an HTTP resource under `http://example.com`.
- ACME runs over HTTPS and message bodies are signed JSON objects.
- The client periodically contacts the server to obtain updated certificates or Online Certificate Status Protocol (OCSP) responses.

Let's Encrypt is a certification authority that provides X.509 certificates at no charge. The certificates are valid of a rather limited time and must be renewed regularly. The certification process is completely automated. Let's Encrypt started to offer certificates officially in April 2016 and on February 2020 one billion certificates were issued.

RFC 8555 [8] defines the protocol that is used by Let's Encrypt to automate the process of verification and certificate issuance. The reference implementation of the server is called `Boulder` and written in Go. A popular client program `certbot` is written in Python and easily integrates with web server software.

The Online Certificate Status Protocol (OCSP) is a lightweight protocol to check whether a certificate has been revoked. It is nowadays often used by web servers in order to provide clients with a recent signed OCSP response such that the client does not have to obtain an OCSP response in a separate interaction with an OCSP server.

Key Exchange Schemes

15 Cryptography Primer

16 Symmetric Encryption Algorithms and Block Ciphers

17 Asymmetric Encryption Algorithms

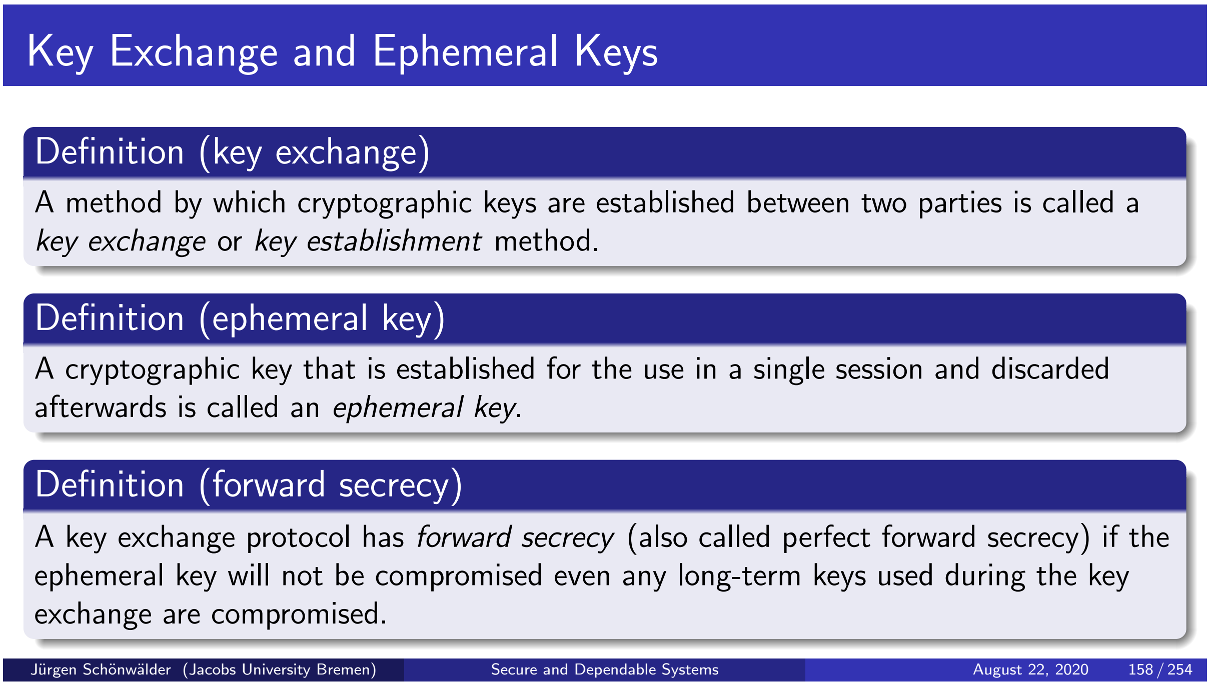
18 Cryptographic Hash Functions

19 Digital Signatures and Certificates

20 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)

A method by which cryptographic keys are established between two parties is called a *key exchange* or *key establishment* method.

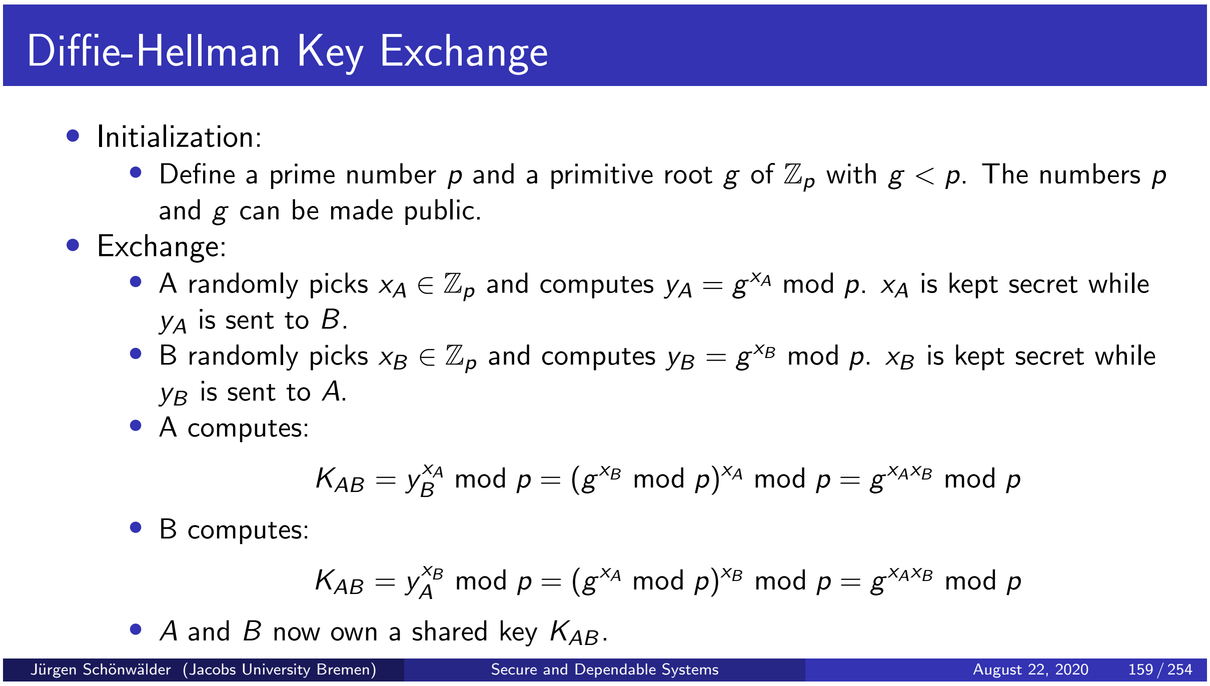
Definition (ephemeral key)

A cryptographic key that is established for the use in a single session and discarded afterwards is called an *ephemeral key*.

Definition (forward secrecy)

A key exchange protocol has *forward secrecy* (also called perfect forward secrecy) if the ephemeral key will not be compromised even any long-term keys used during the key exchange are compromised.

Key exchange methods are widely used to establish ephemeral session keys even if two principals have already access to suitable long-term keys. The reason is that keys can loose their strength the more frequently they are used. Hence, to secure communication over the Internet, it is desirable to establish session keys instead of using long-term keys held by a server directly. If the key exchange mechanism provides (perfect) forward secrecy, then the session keys remain strong even if the server keys get compromised at some point in time 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}.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 159 / 254

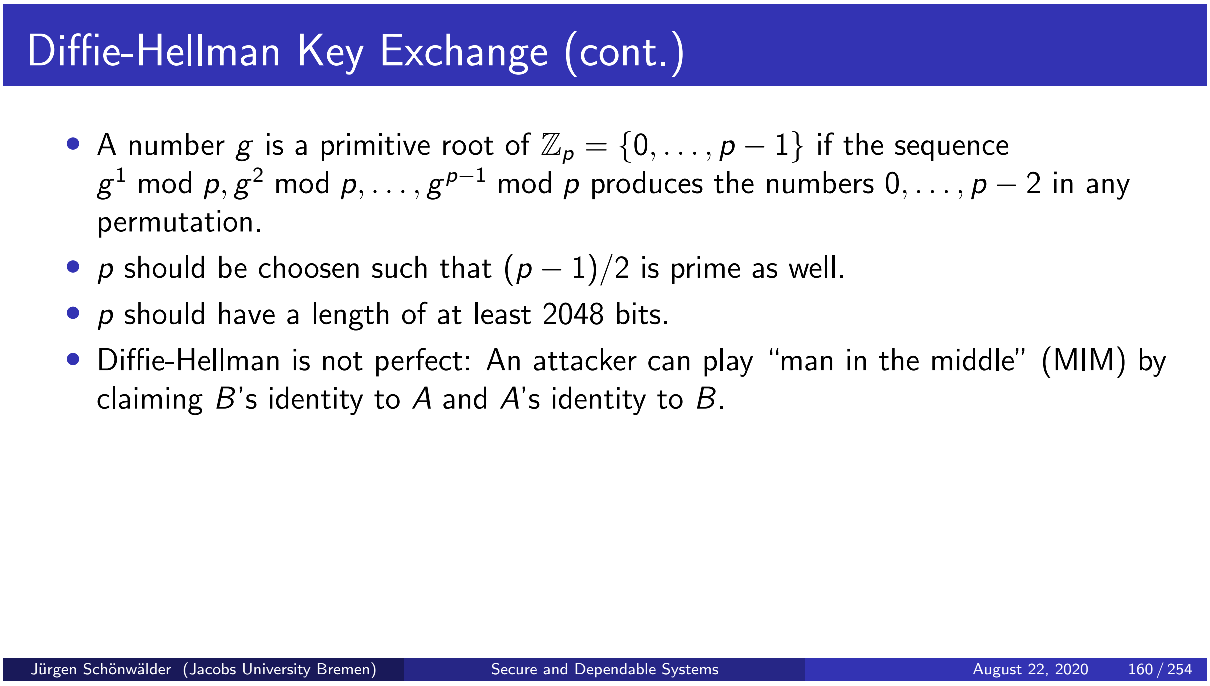
The Diffie-Hellman key exchange [17] uses discrete exponentiation $b^x \bmod m$, which is fast to compute even for large exponents x. For the inverse function, the discrete logarithm, there is no known efficient algorithm (a probabilistic algorithm with polynomial time). The Diffie-Hellman key exchange 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 $t \in \{1, 2, \ldots, p-1\}$ results in the numbers $\{0, 1, \ldots, p-2\}$ (in any order).

```haskell
import Data.List

-- check whether p is a prime number
isPrime :: Integer -> Bool
isPrime p = null [ x | x <- 2:[3,5..r], p `mod` x == 0]
    where r = (floor . sqrt . fromIntegral) p

-- 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 $K_{AB} = y_A^{x_B} \bmod p = 2^{22} \bmod 47 = 24$



Diffie-Hellman Key Exchange (cont.)

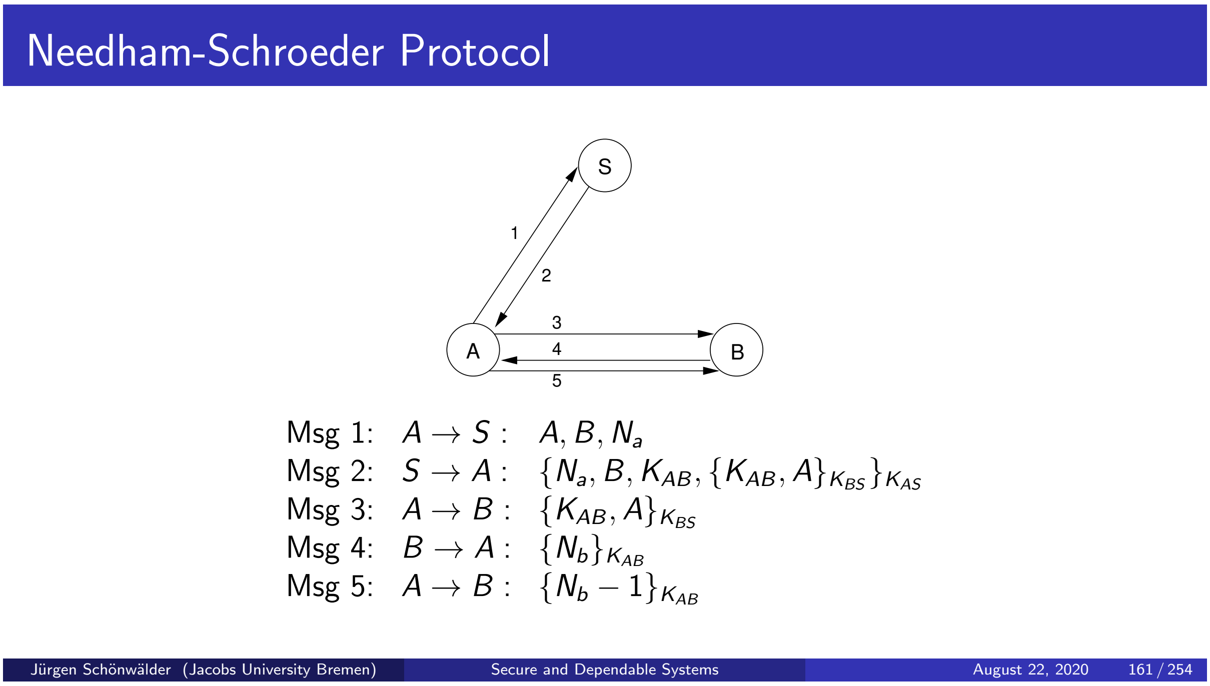
- A number g is a primitive root of $\mathbb{Z}_p = \{0, \ldots, p-1\}$ if the sequence $g^1 \bmod p, g^2 \bmod p, \ldots, g^{p-1} \bmod p$ produces the numbers $0, \ldots, p-2$ in any permutation.
- p should be choosen such that $(p-1)/2$ is prime as well.
- p should have a length of at least 2048 bits.
- 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.

The Diffie-Hellman exchange can be attacked easily if an attacker can act as a man-in-the-middle (MIM). 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).

YouTube: Secret Key Exchange (Diffie-Hellman)

YouTube: Diffie Hellman - the Mathematics bit

YouTube: Explaining the Diffie-Hellman Key Exchange



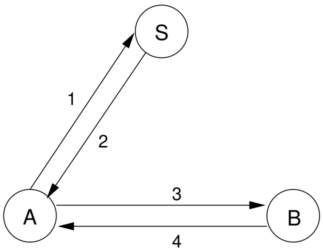
Needham-Schroeder Protocol

$$\begin{array}{lll} \text{Msg 1:} & A \rightarrow S: & A, B, N_a \\ \text{Msg 2:} & S \rightarrow A: & \{N_a, B, K_{AB}, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}} \\ \text{Msg 3:} & A \rightarrow B: & \{K_{AB}, A\}_{K_{BS}} \\ \text{Msg 4:} & B \rightarrow A: & \{N_b\}_{K_{AB}} \\ \text{Msg 5:} & A \rightarrow B: & \{N_b - 1\}_{K_{AB}} \end{array}$$

The Needham-Schroeder protocol [36] assumes that the two principals A and B both share a key with the server S. (This shared key may be derived from a password.)

- 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.

Kerberos Protocol



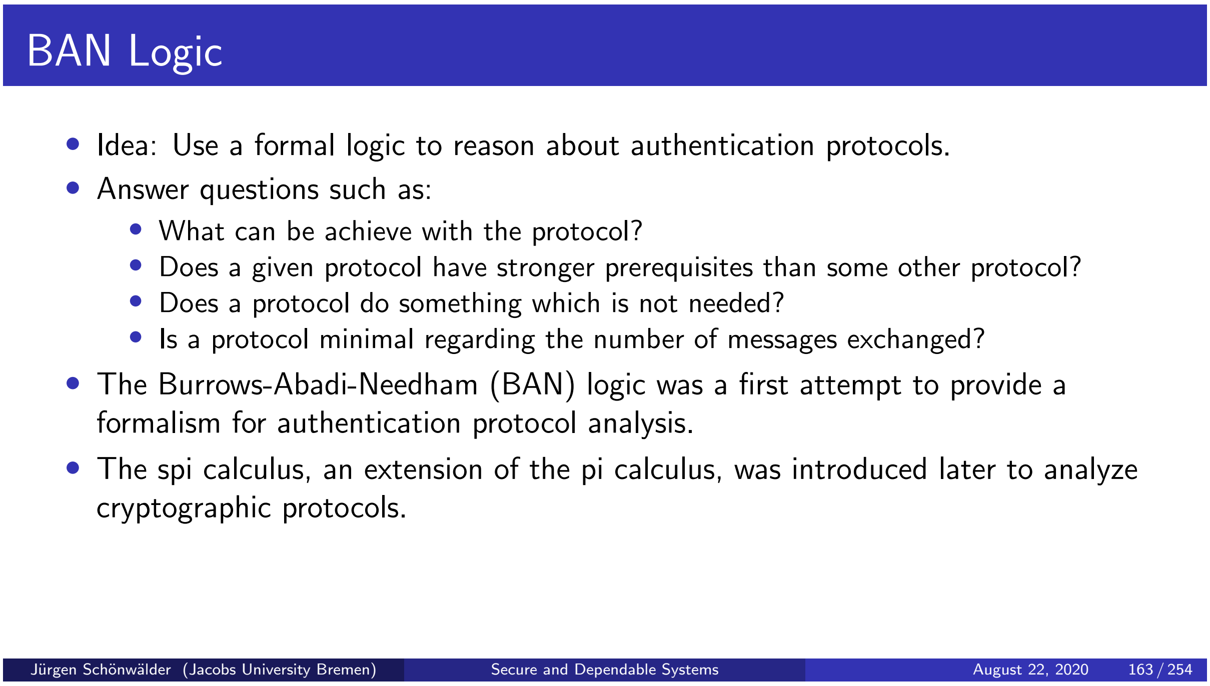
$$\begin{array}{lll}
\text{Msg 1:} & A \rightarrow S: & A, B \\
\text{Msg 2:} & S \rightarrow A: & \{T_s, L, K_{AB}, B, \{T_s, L, K_{AB}, A\}_{K_{BS}}\}_{K_{AS}} \\
\text{Msg 3:} & A \rightarrow B: & \{T_s, L, K_{AB}, A\}_{K_{BS}}, \{A, T_a\}_{K_{AB}} \\
\text{Msg 4:} & B \rightarrow A: & \{T_a + 1\}_{K_{AB}}
\end{array}$$

The Kerberos authentication service was developed at MIT. Version 5 of Kerberos is defined in RFC 4120 [37]. RFC 6649 [5] and RFC 8429 [25] deprecate weak crytographic algorithms in Kerberos.

- This is an 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 clocks, see [27].

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 [11] and the spi calculus about ten years later in 1999 [3]. Formal approaches to prove the correctness of crytographic protocols are important. For a recent example, see the verification of the TLS 1.3 protocol in [15].

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. Remove unnecessary elements 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.

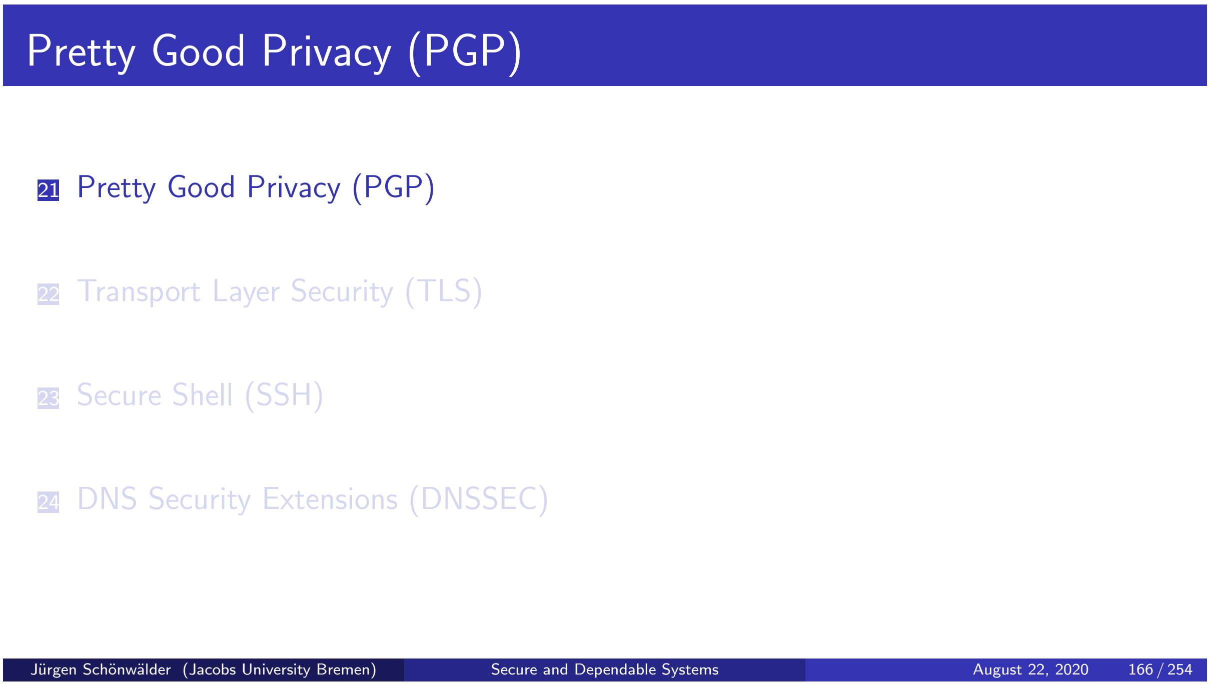
We look at some of the more successful secure protocol designs. People interested in understanding secure protocol designs in more detail should, however, also study the failed designs. There is often quite a bit to learn from design failures.

We will first look at an email security solution (PGP) that can be used in general to protect documents and not just email messages. PGP is, for example, used to secure software distribution or to securely store keys.

We then study transport layer security (TLS), a protocol very widely used to secure communication over the Internet. It was originally developed to secure the World Wide Web in order to enable commerce over the Internet. These days, TLS (and its cousin DTLS) is used to secure many other protocols as well.

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. SSH is often also used to access services such as github or to transfer files or directory tress securely between computers.

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).



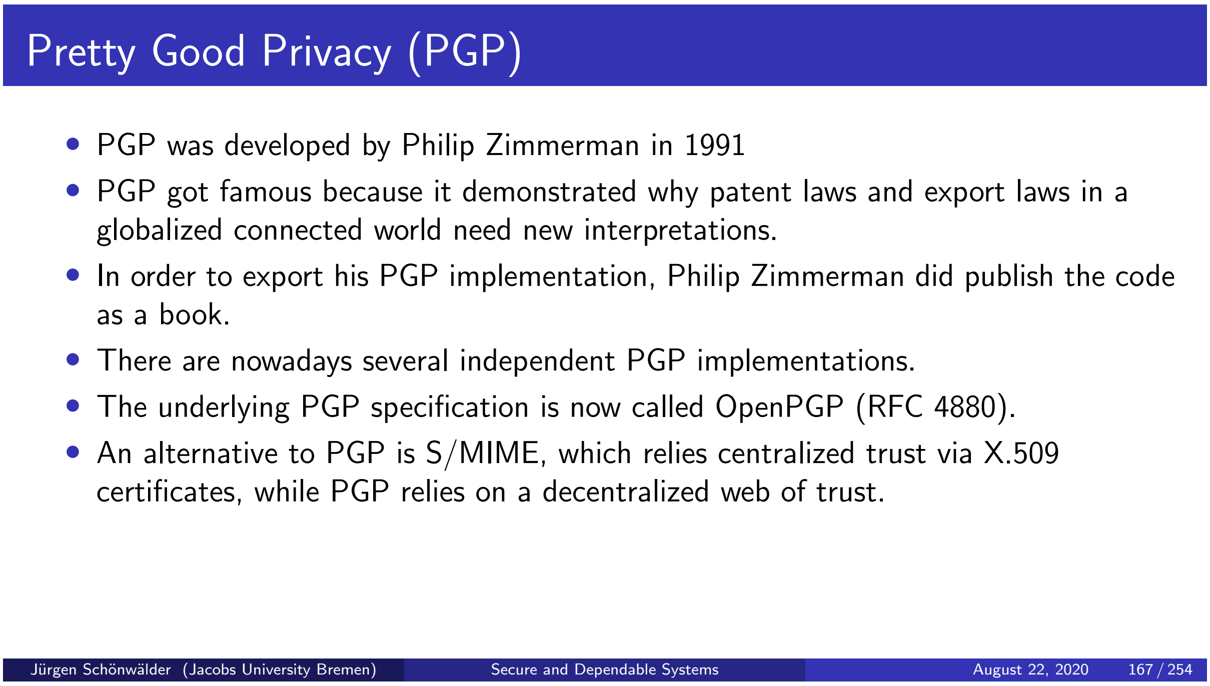
Pretty Good Privacy (PGP)

21 Pretty Good Privacy (PGP)

22 Transport Layer Security (TLS)

23 Secure Shell (SSH)

24 DNS Security Extensions (DNSSEC)



Pretty Good Privacy (PGP)

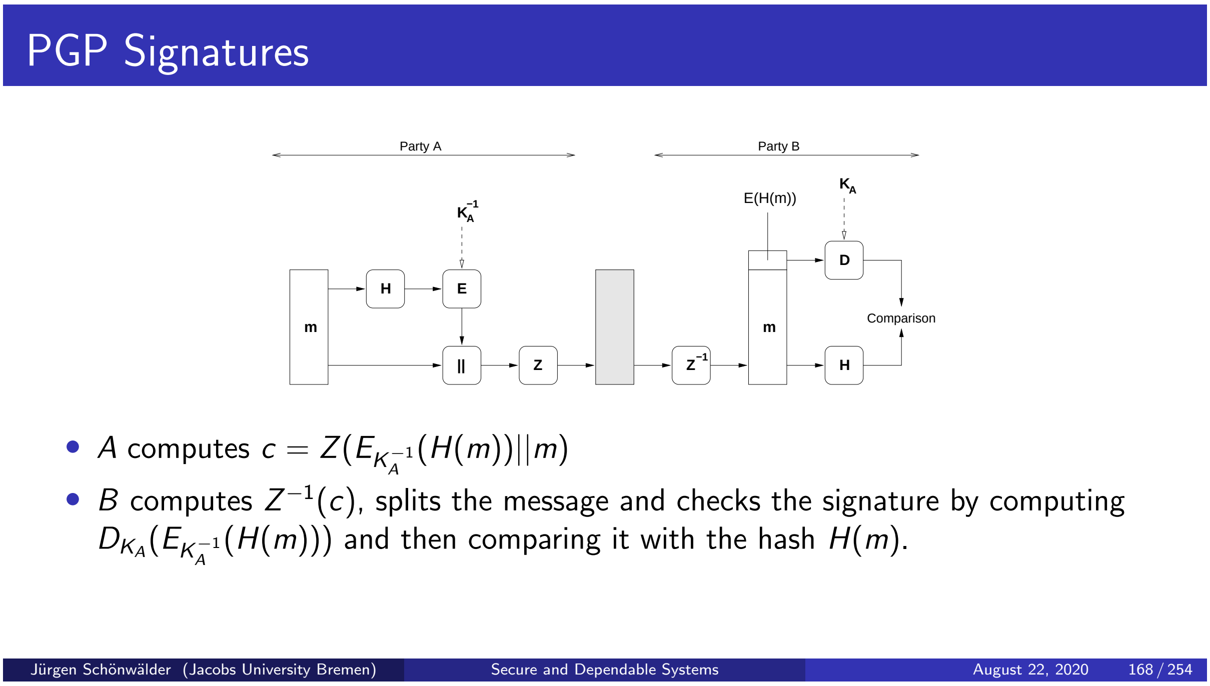
- PGP was developed by Philip Zimmerman in 1991
- PGP got famous because it demonstrated why patent laws and export laws in a globalized connected world need new interpretations.
- In order to export his PGP implementation, Philip Zimmerman did publish the code as a book.
- There are nowadays several independent PGP implementations.
- The underlying PGP specification is now called OpenPGP (RFC 4880).
- An alternative to PGP is S/MIME, which relies centralized trust via X.509 certificates, while PGP relies on a decentralized web of trust.

A popular implementation of RFC 4880 [12] is the Gnu Privacy Guard (`gpg`). We will use `gpg` in the following examples.

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” [10].

YouTube: Stories from the Crypto Revolution



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 comparing it with the hash $H(m)$.

To produce a signed message of a cleartext m

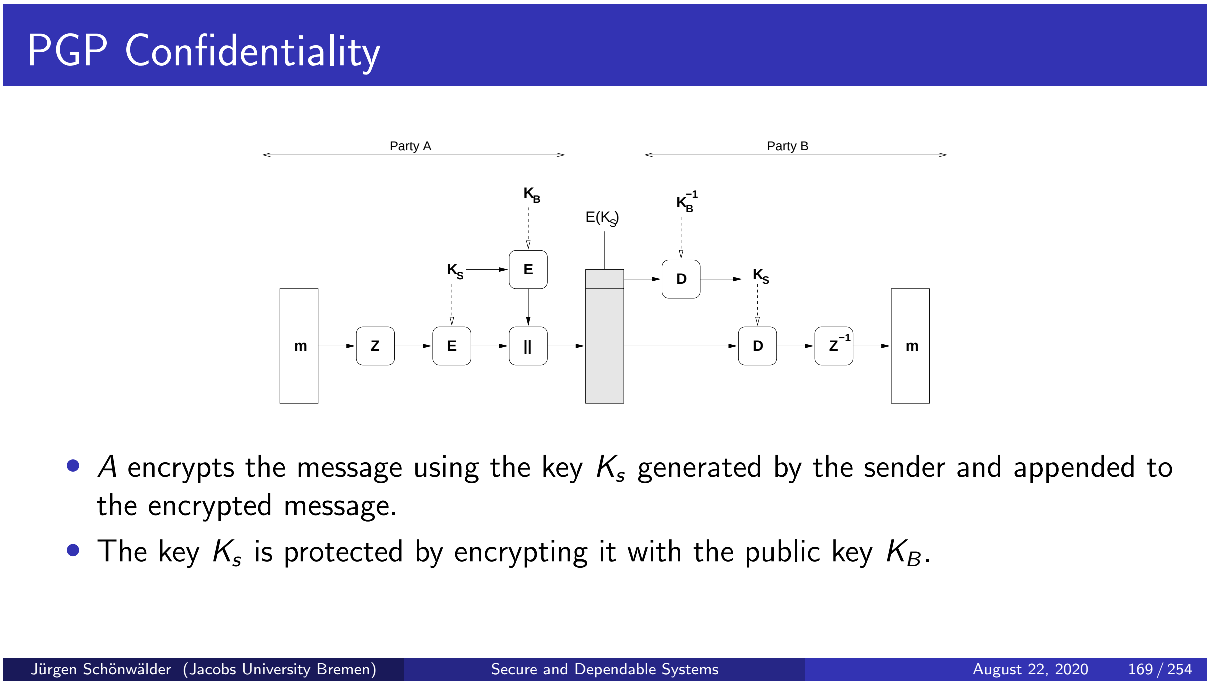
1. a cryptographic hash is computed over m;
2. the hash is encrypted using the signer's private key;
3. the signed hash is appended to the cleartext;
4. the resulting message is compressed.

To verify a signed message c

1. the message c is uncompressed;
2. the message is split into the signature and the cleartext;
3. a cryptographic hash is computed over m;
4. the encrypted hash is decrypted with the signer's public key;
5. the two hash values are compared.

PGP originally used the hash-function MD5, the public-key algorithm RSA and zlib compression. Newer versions support crypto agility. Below is an example how `gpg` can be used to sign a document. Note that `gpg --sign` creates a signature that includes the content of the message. (You can retrieve the content using `gpg --decrypt`.) In order to create a signature that is detached from the document, you have to use `gpg --detach-sign`. The `--armor` option requests to create ASCII armored output, the default is to create the binary OpenPGP format.

```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ gpg --detach-sign --armor --output welcome.txt.sig welcome.txt
$ gpg --verify welcome.txt.sig welcome.txt
gpg: Signature made Tue Apr 14 10:11:07 2020 CEST
gpg:                using RSA key 5C99D6C020BE2520C6F485B8761F2585384508AF
gpg: Good signature from "Juergen Schoenwaelder <j.schoenwaelder@jacobs-university.de>" [ultimate]
$ gpg --sign --armor --output welcome.sig welcome.txt
$ gpg --verify welcome.sig
gpg: Signature made Tue Apr 14 10:02:51 2020 CEST
gpg:                using RSA key 5C99D6C020BE2520C6F485B8761F2585384508AF
gpg: Good signature from "Juergen Schoenwaelder <j.schoenwaelder@jacobs-university.de>" [ultimate]
```



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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 169 / 254

To produce an encrypted message of a cleartext m

1. compress the message m;
2. generate a key K_s;
3. the compressed message is encrypted using the key K_s;
4. the key K_s is encrypted using the receiver's public key;
5. the encrypted key is appended to the encrypted message.

To receive an encrypted message c

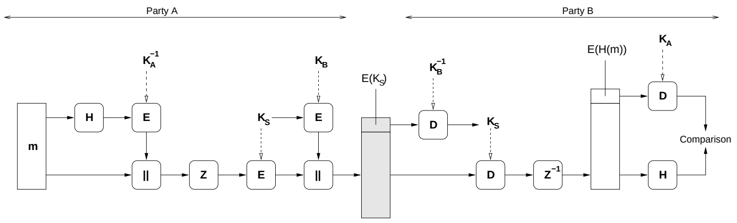
1. the message is split into the encrypted key and the encrypted message;
2. the encrypted key is decrypted using the receiver's private key;
3. the encrypted message is decrypted using K_s;
4. the resulting compressed message is decompressed.

PGP confidentiality combines symmetric encryption algorithms, which are fast on large inputs, with asymmetric encryption algorithms, which make it easy identify communicating parties (and to exchange keys).

Below is an example showing how a document can be encrypted such that only a give recipient can decrypt and read it. The `--symmetric` option can be used to create an encrypted file that can be decrypted by anyone who has the correct passphrase.

```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ gpg --encrypt --output welcome.gpg --recipient j.schoenwaelder@jacobs-university.de welcome.txt
$ gpg --decrypt welcome.gpg
gpg: encrypted with 2048-bit RSA key, ID 6CD3F6BCC2CFB1E2, created 2018-05-09
      "Juergen Schoenwaelder <j.schoenwaelder@jacobs-university.de>"
Welcome to Secure and Dependable Systems
$ gpg --symmetric --output welcome.gpg welcome.txt
$ gpg --decrypt welcome.gpg 2>/dev/null
Welcome to Secure and Dependable Systems
```

PGP Signatures and Confidentiality



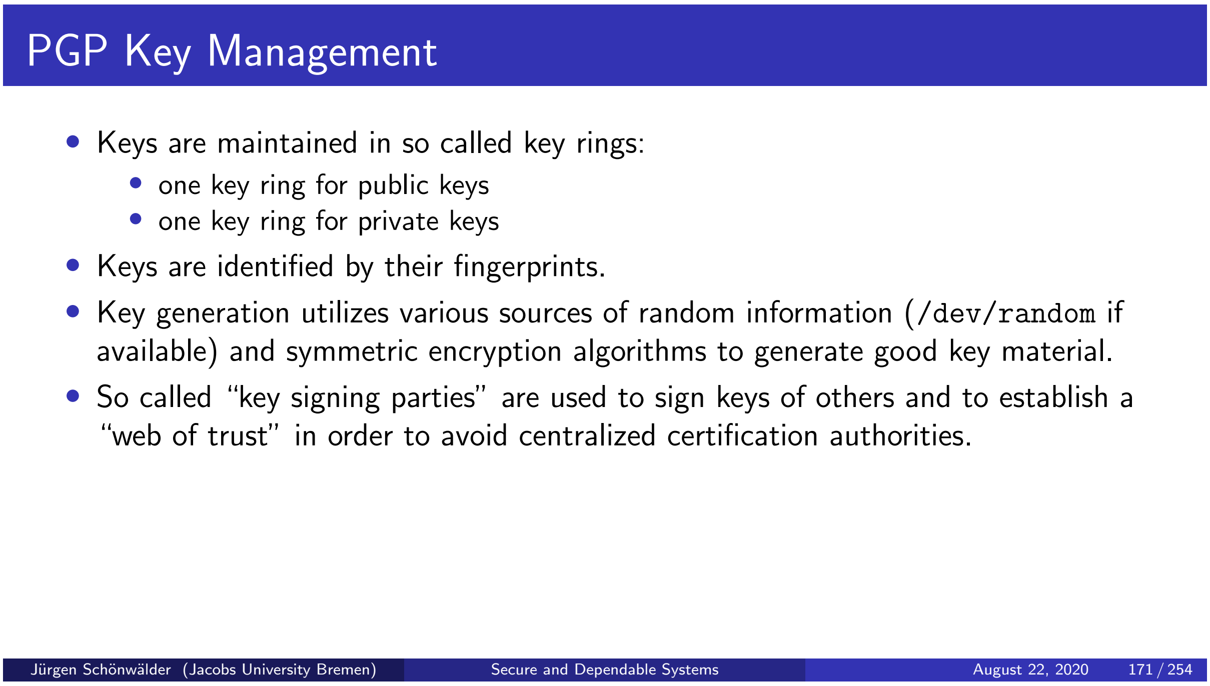
- Signature and confidentiality can be combined as shown above.
- PGP uses in addition Radix-64 encoding (a variant of base-64 encoding) to ensure that messages can be represented using the ASCII character set.
- PGP supports segmentation/reassembly functions for very large messages.

To send an encrypted message with the signature of the sender, the two previous algorithms are combined.

All three schemes require that all communicating parties have access to the public keys and that they trust these keys to belong to the correct identity. Also note that protected messages can have a long lifetime during which (i) keys may simply get lost or (ii) keys may get stolen or broken.

Below is an example showing how a document can be both encrypted and signed.

```
$ echo "Welcome to Secure and Dependable Systems" > welcome.txt
$ gpg --encrypt --sign --output welcome.gpg \
> --recipient j.schoenwaelder@jacobs-university.de welcome.txt
$ gpg --decrypt welcome.gpg
gpg: encrypted with 2048-bit RSA key, ID 6CD3F6BCC2CFB1E2, created 2018-05-09
      "Juergen Schoenwaelder <j.schoenwaelder@jacobs-university.de>"
Welcome to Secure and Dependable Systems
gpg: Signature made Tue Apr 14 10:51:35 2020 CEST
gpg:                using RSA key 5C99D6C020BE2520C6F485B8761F2585384508AF
gpg: Good signature from "Juergen Schoenwaelder <j.schoenwaelder@jacobs-university.de>" [ultimate]
```



PGP Key Management

- Keys are maintained in so called key rings:
  - one key ring for public keys
  - one key ring for private keys
- Keys are identified by their fingerprints.
- 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
  ```

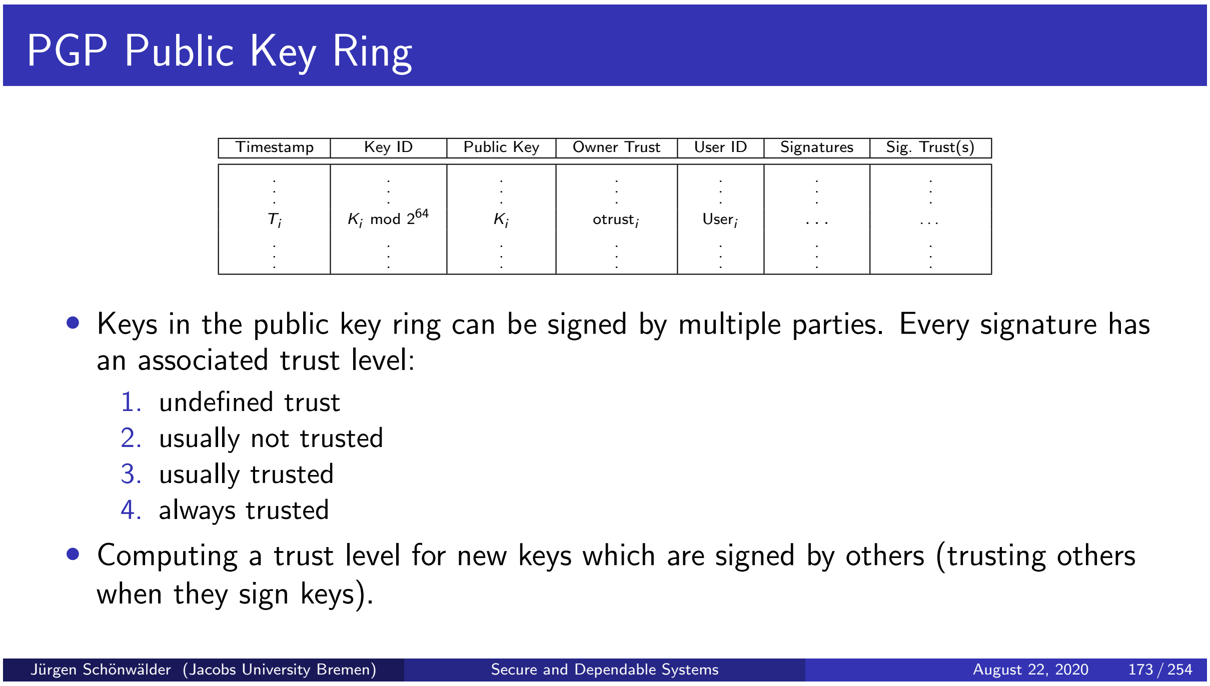
  Send your key with the signatures to a key server:

  ```
  gpg --send-key $MYKEYID
  ```

PGP Private Key Ring

Timestamp	Key ID	Public Key	Encrypted Private Key	User ID
$\vdots$ T_i $\vdots$	$\vdots$ $K_i \bmod 2^{64}$ $\vdots$	$\vdots$ K_i $\vdots$	$\vdots$ $E_{H(P_i)}(K_i^{-1})$ $\vdots$	$\vdots$ User_i $\vdots$

- 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 ⋮	⋮ ... ⋮	⋮ ... ⋮

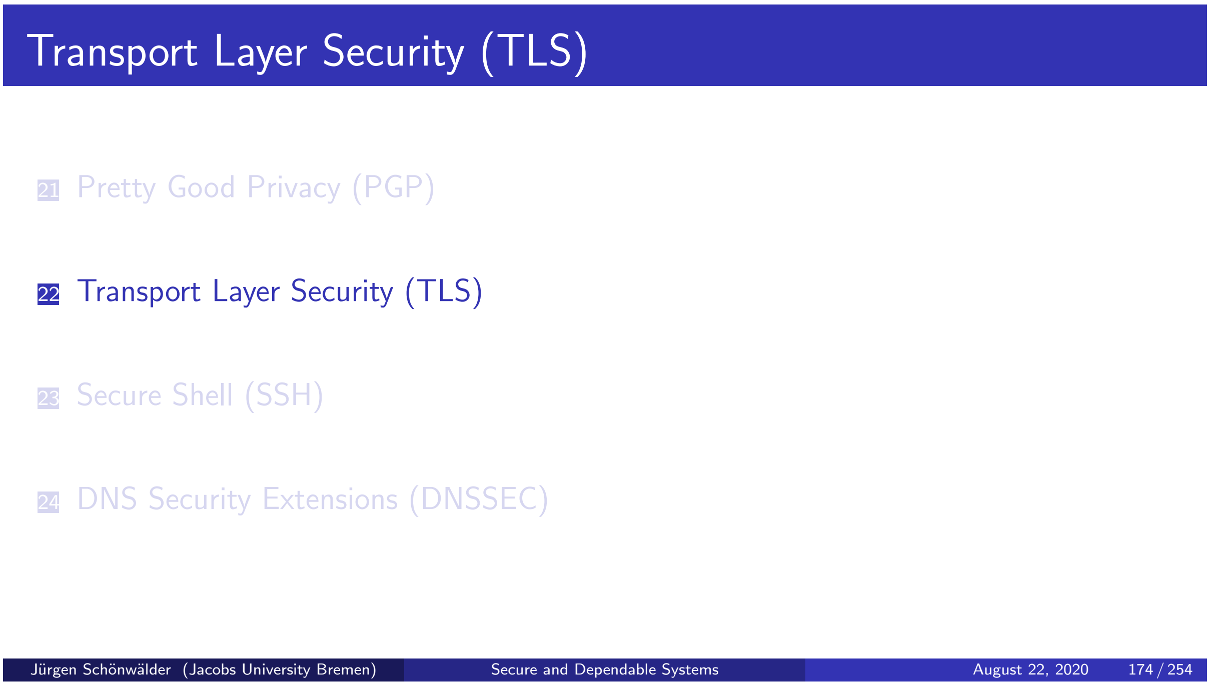
- Keys in the public key ring can be signed by multiple parties. Every signature has an associated trust level:
  1. undefined trust
  2. usually not trusted
  3. usually trusted
  4. always trusted
- Computing a trust level for new keys which are signed by others (trusting others when they sign keys).

The Web of Trust implementation, and in particular the Synchronizing Key Servers (SKS), face several problems:

- Some people submit fake keys. Since there are people who do not carefully check key signatures, these people may end up happily working with fake keys. Fake key spamming is a problem as there is not way to keep the data on the keyservers clean.
- The keys on the SKS keyservers are public and they do include somewhat sensitive information such as email addresses.
- Attacks are possible such as adding a large number of signatures to a key, which then causes failures of programs not prepared to handle keys with many signatures (see the certificate spamming attacks reporting in June 2019 on the SKS keyserver network).

The new `keys.openpgp.org` keyserver enables owners of keys control over their keys and the sensitive identity information associated with their keys. Note that `keys.openpgp.org` does not federate with the SKS pool.

The `keys.openpgp.org` keyserver distributes exactly one current key for an email address and it does not distribute signatures, i.e., the trust into the keys must be obtained via other mechanisms. It is designed to be a pure key lookup mechanism.



Transport Layer Security (TLS)

21 Pretty Good Privacy (PGP)

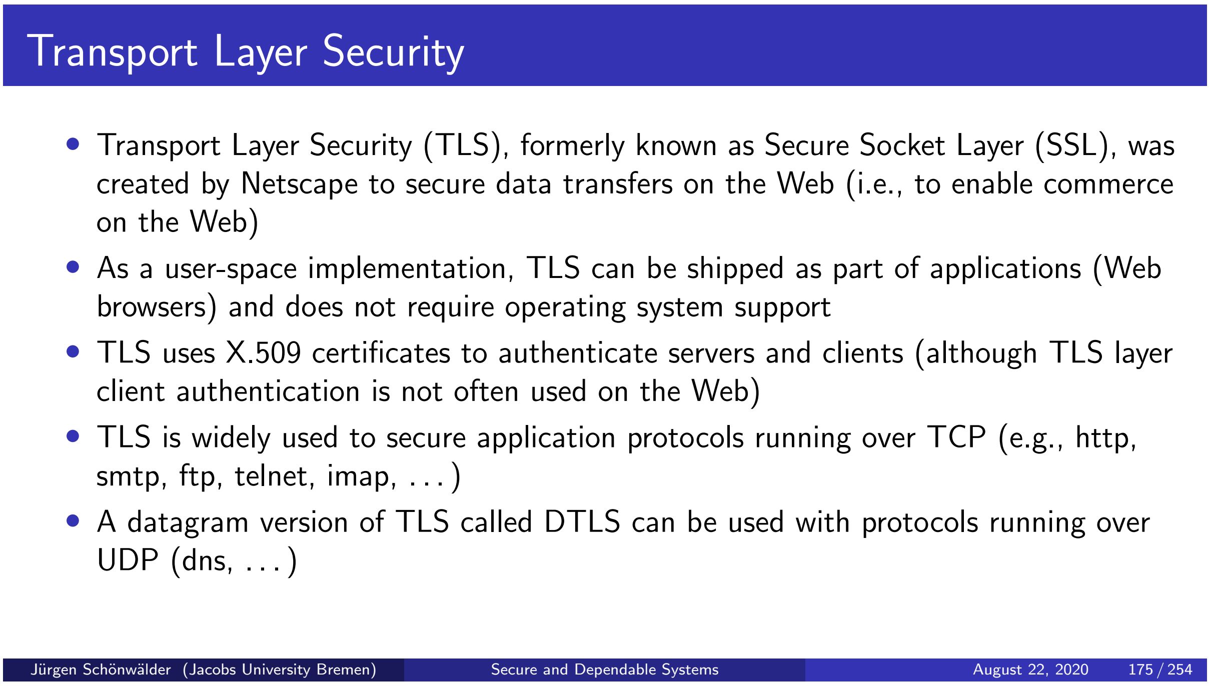
22 Transport Layer Security (TLS)

23 Secure Shell (SSH)

24 DNS Security Extensions (DNSSEC)

Tranport Layer Security (TLS) is probably the most widely used protocol to secure communication over the Internet. Originally designed to enable commerce over the Internet, it is used meanwhile for many other purposes, such as creating secure virtual private networks, to securely access email message stores, or to protect domain name lookups. Initially, the focus was often limited on protecting financial transactions or user authentification dialogues while meanwhile the goal is often to protect the privacy of users communcating over the Internet. (The primary reason to encrypt domain name lookups is to prevent third parties from observing which web sites a user is visiting.)

Security protocols like TLS and their implementations attract of course a lot of people interested in finding weaknesses. These have ranged from protocol flaws (e.g., the TLS renegotiation attack, 2009) or implementation errors (e.g., the OpenSSL heartbleed bug, 2014). Several of the classic attacks known by 2014 have been documented in RFC 7457 [45].

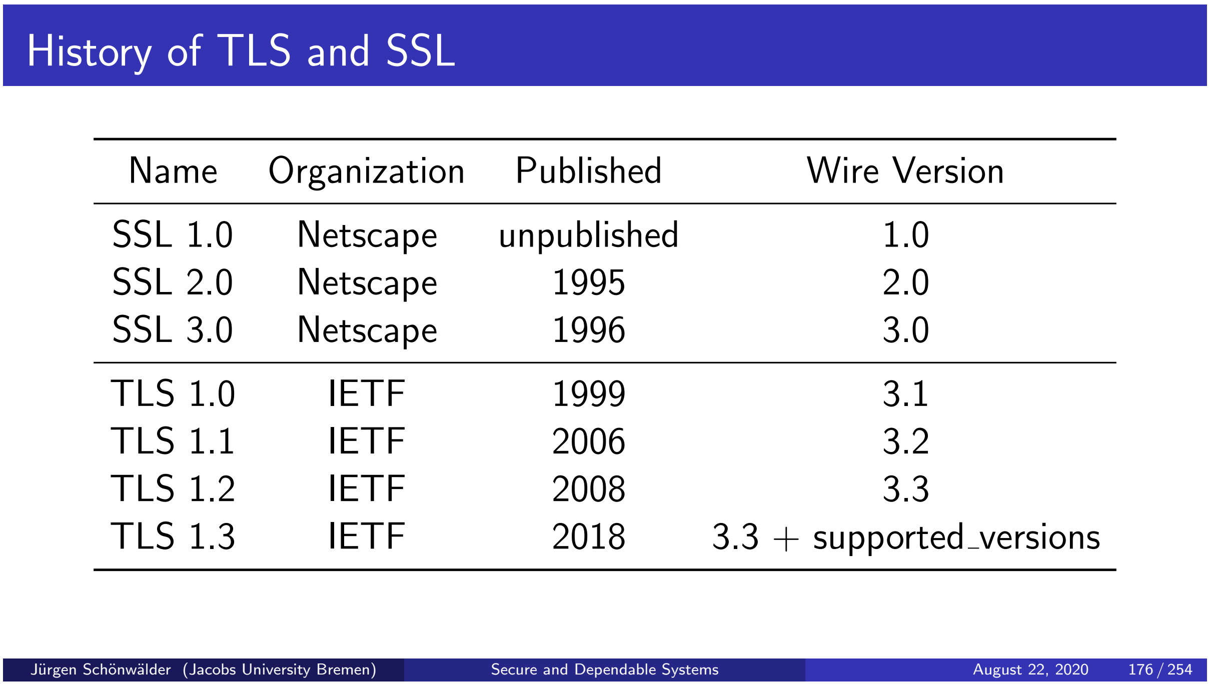


Transport Layer Security

- 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 on the Web)
- 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 (dns, . . .)

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 175 / 254

TLS 1.2 is defined in RFC 5246 [16] and TLS 1.3 has been published recently in RFC 8446 [40]. DTLS 1.2 is defined in RFC 6347 [41]. A good article describing the design of DTLS is [35]. At the point of this writing, TLS versions older then TLS 1.2 are not recommended to be used anymore and implementations started to disable them.

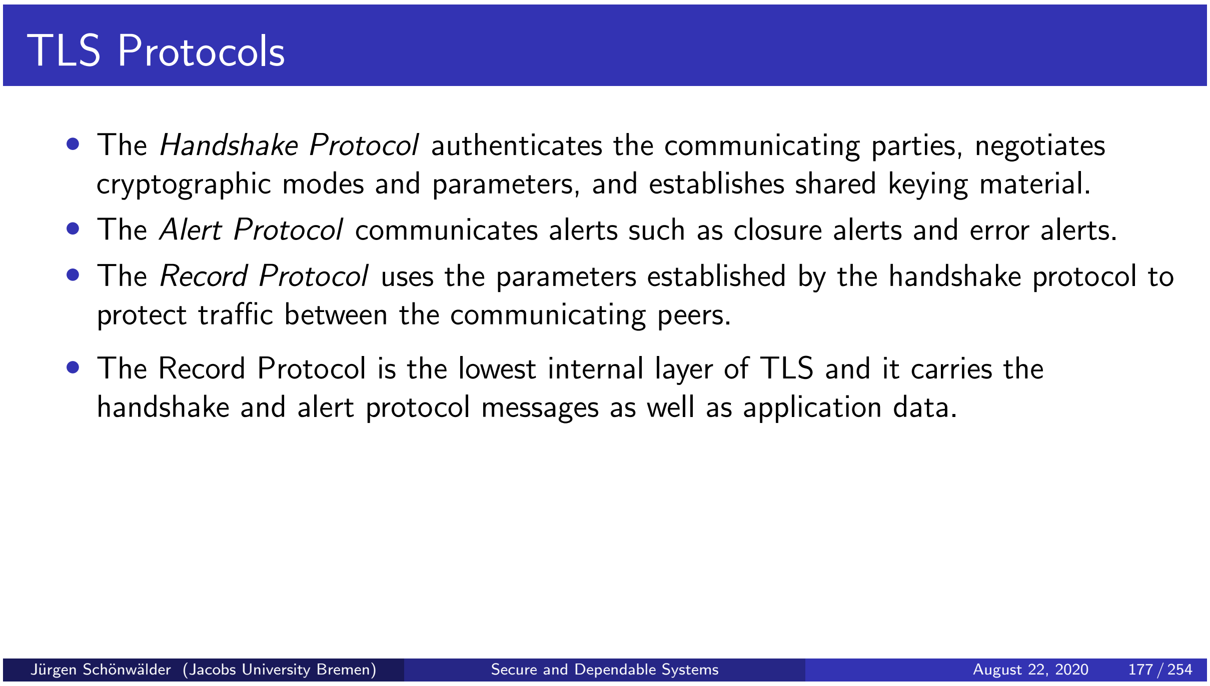


History of TLS and SSL

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

All TLS versions prior to TLS 1.2 are considered outdated at the time of this writing (April 2020). Web browsers are moving towards disabling support for outdated TLS versions, which forces all sites still running old versions of TLS to upgrade to at least TLS 1.2. However, due to the Corona virus outbreak in Spring 2020 and many government sites supporting only outdated versions of TLS, web browsers started to postpone the disabling of old versions of TLS. So it may take longer to stop the usage of outdated TLS versions on the Web.

Attacks on TLS 1.2 have been increasing in recent years. TLS 1.3, published in 2018, introduces a radically different handshake protocol and it removes a large collection of problematic constructs and outdated encryption algorithms. However, only future will tell whether TLS 1.3 is robust to attacks.



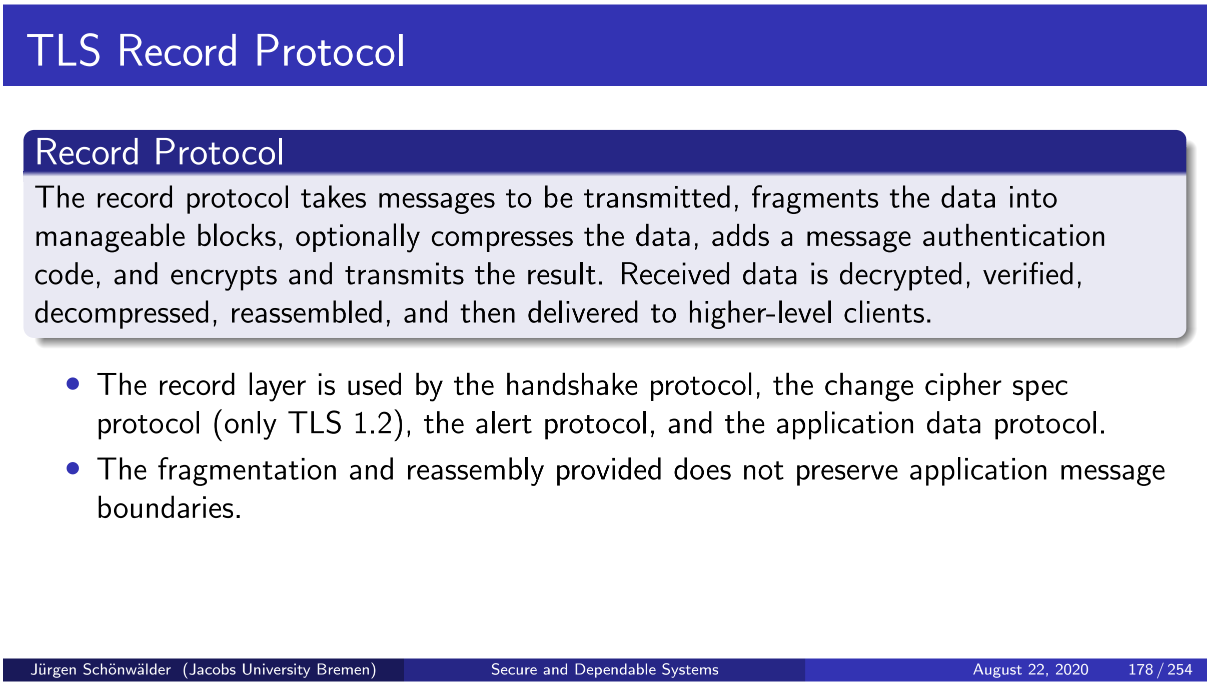
TLS Protocols

- 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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 177 / 254

YouTube: SSL/TLS - Cristina Formaini

YouTube: Stanford Seminar - The TLS 1.3 Protocol



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.

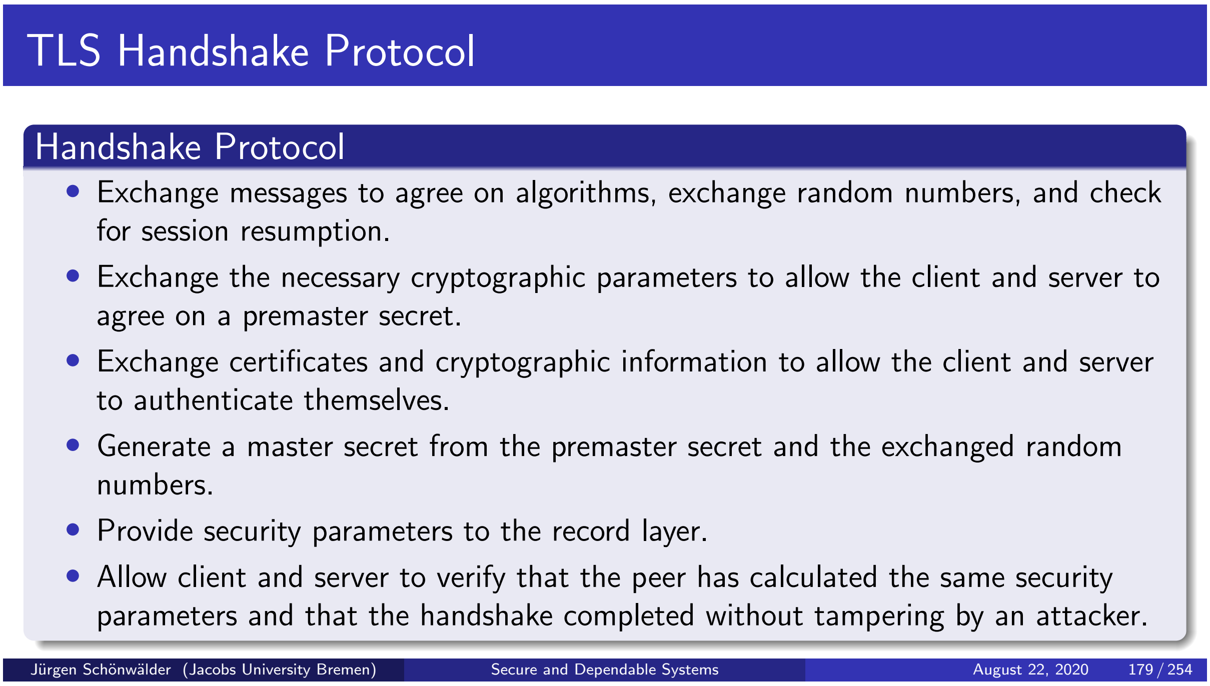
TLS defines message formats using a notation that resembles C. Here is the definition of the record protocol of TLS 1.2 [16]. Note that tha record can be either `TLSPlaintext`, `TLSCompressed`, or `TLSCiphertext`, where the `TLSCiphertext` supports multiple cipher types.

```
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 [40] 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;
```

`TLSCiphertext.encrypted_record` holds the AEAD-encrypted form of the serialized `TLSInnerPlaintext` structure and the `TLSInnerPlaintext.content` holds the `TLSPlaintext.fragment` value, containing the byte encoding of a handshake or an alert message, or the raw bytes of the application's data to send.



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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 179 / 254

A full TLS 1.2 [16] 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.

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data
```

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.

```
      Client                                                Server

      ClientHello                   -------->
                                                       ServerHello
                                                [ChangeCipherSpec]
                                    <--------             Finished
      [ChangeCipherSpec]
      Finished                      -------->
      Application Data              <------->     Application Data
```

TLS 1.2 session resumption requires only one round-trip and saves CPU intensive asymmetric crypto operations. While CPUs on regular servers nowadays can do cryptographic operations reasonably faster, the session resumption benefit has moved to lower latency due to reduced network delays.

TLS 1.3 [40] uses a very different handshake protocol. The full TLS 1.3 handshake looks like this:

```
       Client                                               Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*         -------->
                                                       ServerHello  ^ Key
                                                      + key_share*  | Exch
                                                 + pre_shared_key*  v
                                             {EncryptedExtensions}  ^  Server
                                             {CertificateRequest*}  v  Params
                                                    {Certificate*}  ^
                                              {CertificateVerify*}  | Auth
                                                        {Finished}  v
                                 <--------     [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}                -------->
       [Application Data]        <------->      [Application Data]
```

TLS 1.3 supports a so-called 0-rtt (zero round-trip) mode:

```
            Client                                               Server

            ClientHello
            + early_data
            + key_share*
            + psk_key_exchange_modes
            + pre_shared_key
            (Application Data*)     -------->
                                                            ServerHello
                                                       + pre_shared_key
                                                           + key_share*
                                                  {EncryptedExtensions}
                                                          + early_data*
                                                             {Finished}
                                    <--------       [Application Data*]
            (EndOfEarlyData)
            {Finished}              -------->
            [Application Data]      <------->        [Application Data]
```

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.

It is important that both parties involved in a TLS session properly terminate the session. The alert protocol can be used to signal to the remote party that no more data follows. Note that TLS close notification allows to determine which of the two communicating parties initiates the teardown of the underlying TCP connection. This is important since the party initiating the TCP connection teardown ends up in TCP's `TIME_WAIT` state and in order to be able to scale servers, it is best if the clients initiate the TCP connection teardown.

Secure Shell (SSH)

21 Pretty Good Privacy (PGP)

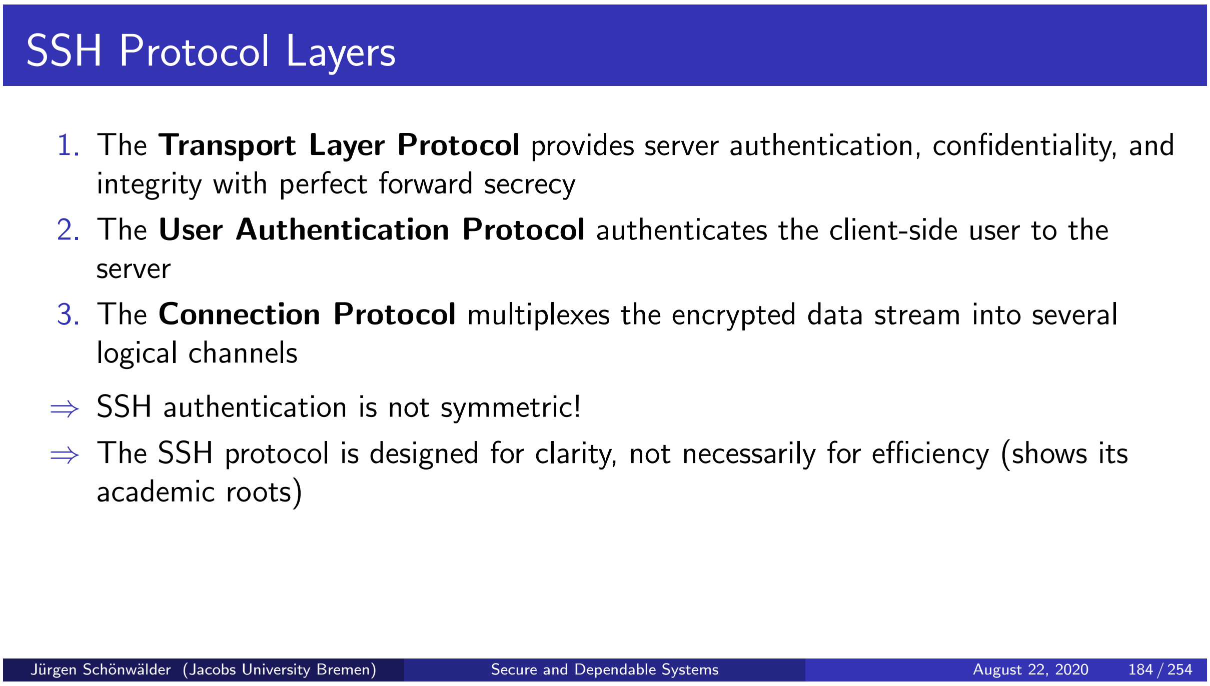
22 Transport Layer Security (TLS)

23 Secure Shell (SSH)

24 DNS Security Extensions (DNSSEC)

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!

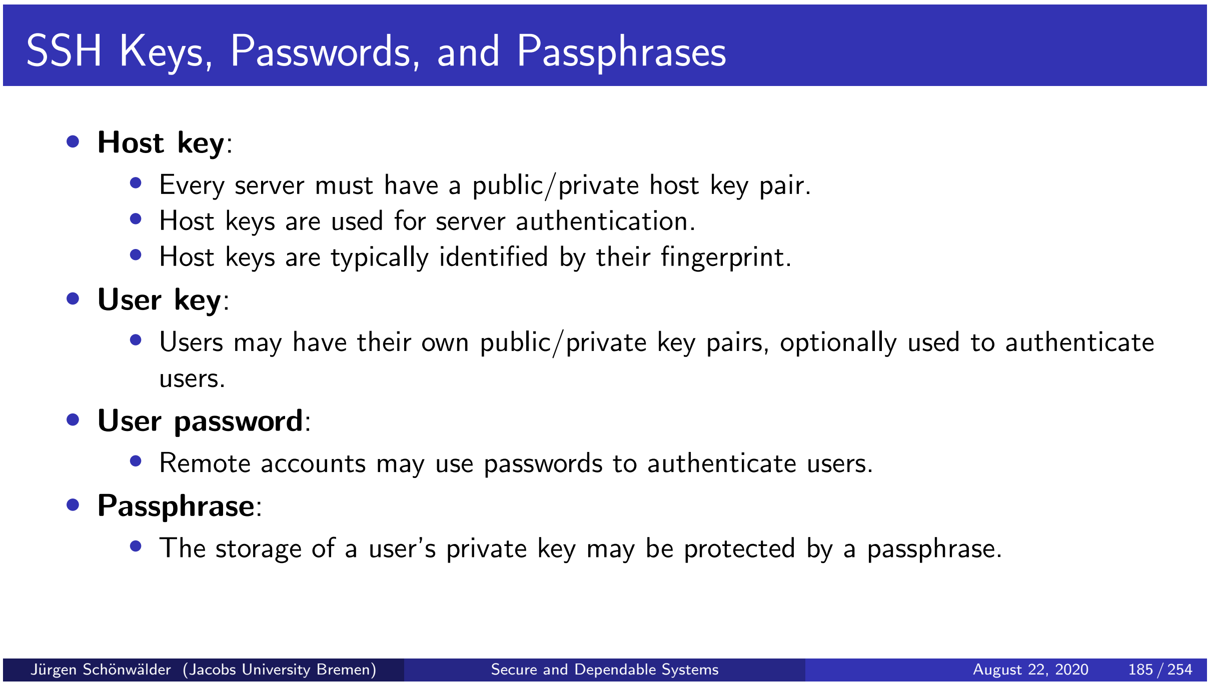
⇒ The SSH protocol is designed for clarity, not necessarily for efficiency (shows its academic roots)

The SSH protocol architecture is defined in RFC 4251 [53]. The SSH transport protocol is defined in RFC 4253 [54] and the user authentication protocol in RFC 4252 [51]. RFC 4254 [52] defines the connection protocol.

YouTube: How SSH Works

YouTube: How Secure Shell Works (SSH) - Computerphile

YouTube: SSH (recorded lecture)



SSH Keys, Passwords, and Passphrases

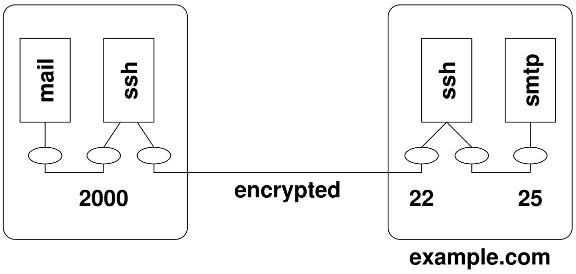
- **Host key**:
  - Every server must have a public/private host key pair.
  - Host keys are used for server authentication.
  - Host keys are typically identified by their fingerprint.
- **User key**:
  - Users may have their own public/private key pairs, optionally used to authenticate users.
- **User password**:
  - Remote accounts may use passwords to authenticate users.
- **Passphrase**:
  - The storage of a user's private key may be protected by a passphrase.

It is important to distinguish between the server's key (called the host key) and user's keys (called the user key). Note that the user authentication protocol can authenticate user's by their keys but also via other means such as simple traditional passwords.

SSH often relies on leap-of-faith to built trust into a host key. When a user connects to a server for the first time, the server asks the user to verify the server's host key fingerprint. In an ideal world, the user would use an out-of-band mechanism to verify that the fingerprint is correct, i.e., that the user to connected to the right server. In reality, user's often blindly accept the host key offered at the first connection time. The SSH client, however, will cache the fingerprint and verify it on every subsequent connection. If the user is happy with the behavior of the remote system, then over time the user builds trust that the fingerprint is the correct one.

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.

Port forwarding can be used to tunnel a protocol over SSH. In the example

```
$ ssh -f joe@example.com -L 2000:example.com:25 -N &
$ nc localhost 1234
220 peach.eecs.jacobs-university.de ESMTP Postfix (Debian/GNU)
```

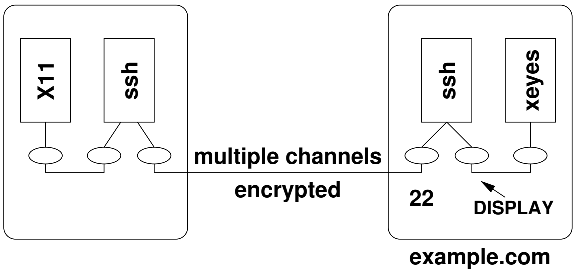
an SSH connection from Joe's local machine to `example.com` using the remote account `joe` is established. The SSH server then creates a tunnel an connects as a client to port `25` on `example.com` and it provides a listening endpoint on the Joe's local host on port 2000. Thus, if a program on Joe's local host connects to the local port 2000, it actually talks to the server on `example.com` using port `25`.

Note that in the example anybody can connect to Joe's local computer on port `1234` to reach the mail server. This can be prevented by restricting access to port `1234` to Joe's local computer.

```
$ ssh -f joe@example.com -L localhost:2000:example.com:25 -N &
```

SSH Features: X11 Forwarding

```
ssh -X joe@example.com
```



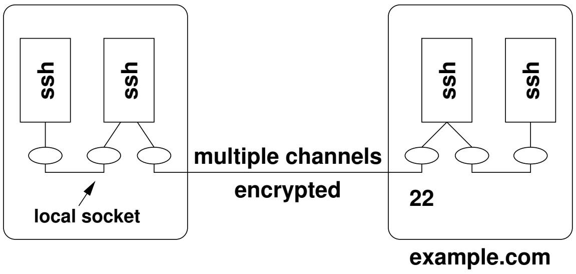
- X11 forwarding is a special application of TCP forwarding allowing X11 clients on remote machines to access the local X11 server (managing the display and the keyboard/mouse).

A typical use case for TCP port forwarding (and likely the reason for inventing it in the first place) is the traditional X window system. The X Window System consists of an X server controlling the display, the keyboard, the pointing device etc. Applications providing a graphical user interface incorporate X clients that connect to an X server in order to draw on the display and to receive events from the X server. The X11 protocol details the information flow between an X server and connected X clients.

SSH's TCP port forwarding mechanism can be used to run graphical applications (X clients) on remote computers that present their graphical user interface on a local machine (via the local X server).

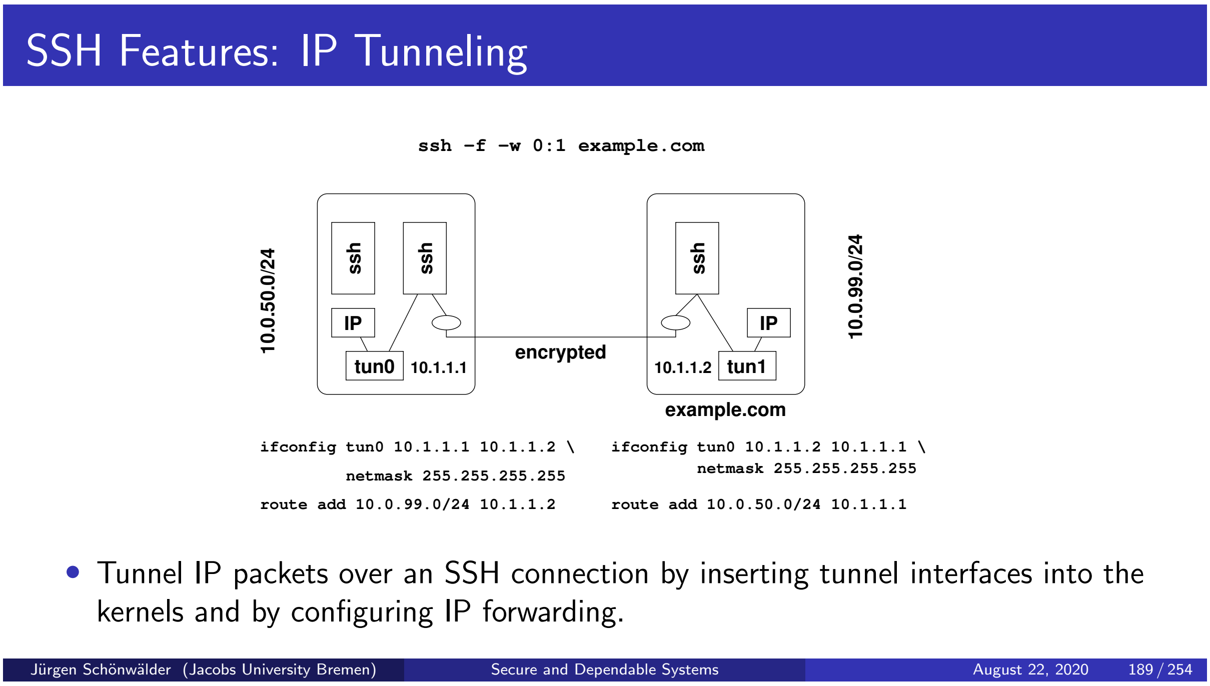
SSH Features: Connection Sharing

`ssh joe@example.com`



- 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).

Since SSH can multiplex multiple channels, it is possible to create new SSH connections on top of existing connections. The benefit of this is greatly reduced connection startup time since no new security context needs to be established. Of course, as a slight downside, the connections share fate. That said, connection sharing provides a significant performance improvements in situations where many otherwise short-lived connections would be used. Examples are system administration tools like ansible.



SSH Features: IP Tunneling

```
ssh -f -w 0:1 example.com
```

```
ifconfig tun0 10.1.1.1 10.1.1.2 \
         netmask 255.255.255.255
route add 10.0.99.0/24 10.1.1.2
```

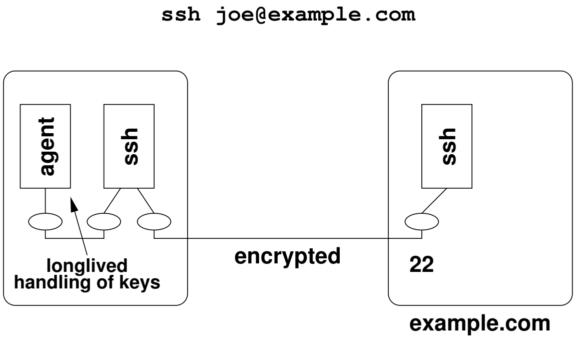
```
ifconfig tun0 10.1.1.2 10.1.1.1 \
         netmask 255.255.255.255
route add 10.0.50.0/24 10.1.1.1
```

- Tunnel IP packets over an SSH connection by inserting tunnel interfaces into the kernels and by configuring IP forwarding.

SSH tunneling generalizes SSH port forwarding even further. Instead of forwarding transport layer connections, SSH is now tunneling network layer IP packets. This essentially provides you with a simple virtual private network (VPN) solution over which you can securely send arbitrary IP traffic. Doing this, however, requires the permissions on both systems to create tunnel network interfaces and to configure IP layer routing as desired.

System administrators and network operators may also consider such tunnels as ways to create backdoors into a network and they may prevent the usage of SSH in IP tunneling mode.

SSH Features: SSH Agent

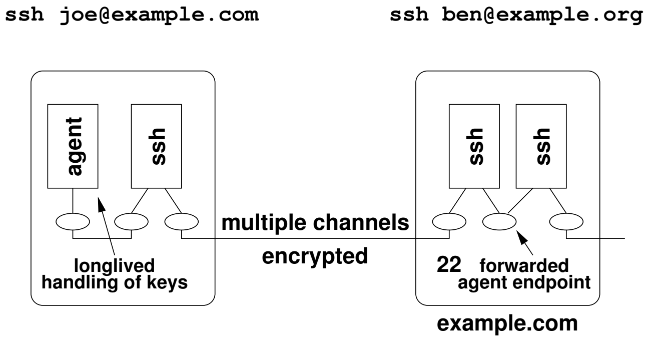


- Maintains client credentials during a login session so that credentials can be reused by different SSH invocations without further user interaction.

SSH user keys are usually stored in files that are encrypted using a key derived from a passphrase. When access to a user key is needed, the user is prompted for the passphrase. In order to reduce the number of times a user has to enter the passphrase, the SSH agent may store decrypted user keys in memory for a certain period of time.

From a user's perspective, when an SSH connection is created, the SSH client tries to talk to the local SSH agent in order to obtain the user's keys. It will fallback to ask the user for a passphrase in case the local SSH agent does not hold the user key or is not accessible. A common approach is to start an ssh-agent when a user starts a login session and to load the user keys when the first SSH connection is established. In such a setup, it is crucial that the SSH agent is terminated when the user's session ends (since otherwise decrypted keys stay around in memory).

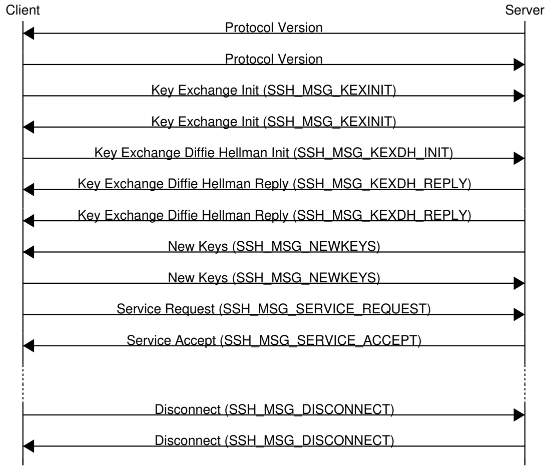
SSH Features: SSH Agent Forwarding



- An SSH server emulates an SSH Agent and forwards requests to the SSH Agent of its client, creating a chain of SSH Agent delegations.

While a local SSH agent is already very convenient, it is possible to go one step further by forwarding the port providing access to the SSH agent to a remote system. This setup enables the user to access a remote system and from there to access further systems, always accessing the local SSH agent. This way, SSH connections can go over multiple hops in a very convenient way without having to store any user keys on intermediate systems.

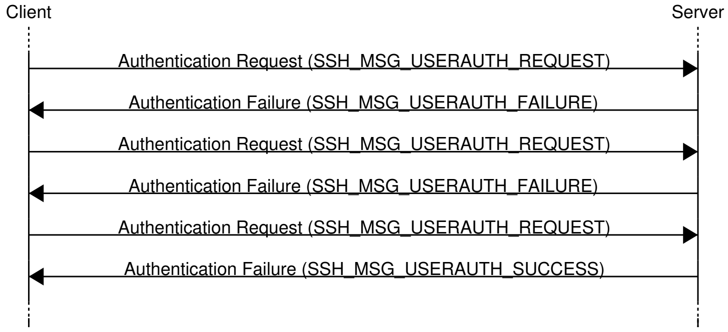
SSH Transport Protocol


Client
Server
Protocol Version
Protocol Version
Key Exchange Init (SSH_MSG_KEXINIT)
Key Exchange Init (SSH_MSG_KEXINIT)
Key Exchange Diffie Hellman Init (SSH_MSG_KEXDH_INIT)
Key Exchange Diffie Hellman Reply (SSH_MSG_KEXDH_REPLY)
Key Exchange Diffie Hellman Reply (SSH_MSG_KEXDH_REPLY)
New Keys (SSH_MSG_NEWKEYS)
New Keys (SSH_MSG_NEWKEYS)
Service Request (SSH_MSG_SERVICE_REQUEST)
Service Accept (SSH_MSG_SERVICE_ACCEPT)
Disconnect (SSH_MSG_DISCONNECT)
Disconnect (SSH_MSG_DISCONNECT)

The transport protocol (RFC 4253) provides strong encryption, server authentication, integrity protection, and optionally compression. The transport protocol typically runs over TCP. The key exchange protocol does automatic key re-exchange, usually after 1 GB of data have been transferred or after 1 hour has passed, whichever is sooner. The cryptographic primitives and the key exchange mechanisms have been extended several times since the publication of RFC 4253.

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. There are differnet key exchange algorithms such as Diffie-Hellman style key exchange or GSS-API style key exchange as well as different host key algorithms. The Host key is used to authenticate the key exchange, to ensure that the client establishes a session key with the correct server.

SSH User Authentication


Client
Server
Authentication Request (SSH_MSG_USERAUTH_REQUEST)
Authentication Failure (SSH_MSG_USERAUTH_FAILURE)
Authentication Request (SSH_MSG_USERAUTH_REQUEST)
Authentication Failure (SSH_MSG_USERAUTH_FAILURE)
Authentication Request (SSH_MSG_USERAUTH_REQUEST)
Authentication Failure (SSH_MSG_USERAUTH_SUCCESS)

- The user authentication protocol iterates through a list of mechanisms until either authentication was successful or all mechanisms have failed.

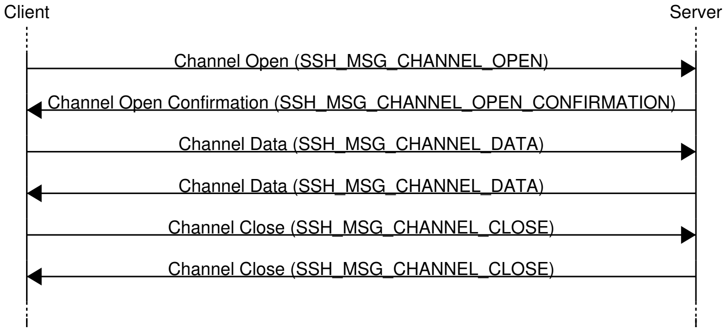
Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 193 / 254

The user authentication protocol (RFC 4252) executes after transport protocol initialization (key exchange) to authenticate the client to the server. There are several authentication methods and the set of methods can be extended:

- 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 (traditional certificates)

Note that user authentication is client driven.

SSH Connection Protocol


Client
Server
Channel Open (SSH_MSG_CHANNEL_OPEN)
Channel Open Confirmation (SSH_MSG_CHANNEL_OPEN_CONFIRMATION)
Channel Data (SSH_MSG_CHANNEL_DATA)
Channel Data (SSH_MSG_CHANNEL_DATA)
Channel Close (SSH_MSG_CHANNEL_CLOSE)
Channel Close (SSH_MSG_CHANNEL_CLOSE)

- The connection protocol has additional messages to handle control flow, error messages (equivalent of stderr), and end-of-file indicators.

The connection protocol (RFC 4254) allows clients to open multiple independent channels. All channels are multiplexed over a single secure SSH transport.

- Channel requests are used to relay out-of-band channel specific data (e.g., window resizing information).
- Channels are widely used for TCP forwarding.

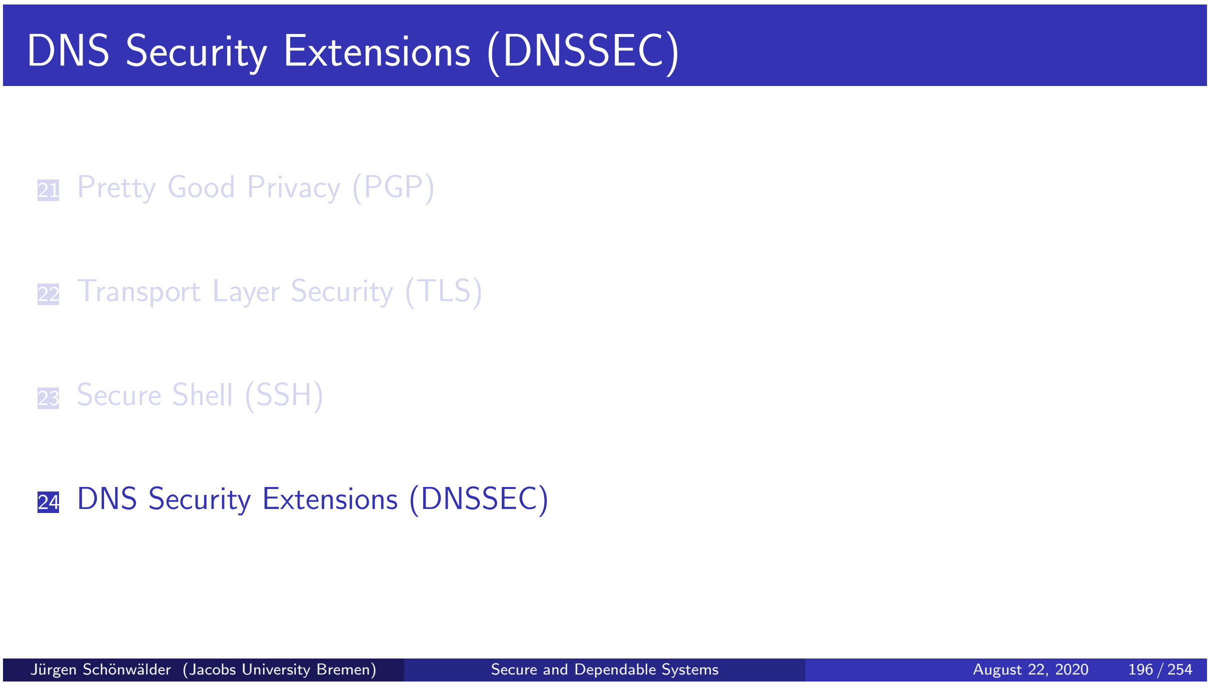
OpenSSH Privilege Separation

- 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.

The widely used OpenSSH implementation uses privilege separation to minimize the amount of code executed with special privileges. While privilege separation has some implementation costs, it is a very effective way to improve the security of complex server software. You can see privilege at work when you access a remote system and look at the process tree. You usually find something like this:

```
`-sshd -D
    `-sshd
        `-sshd,joe
            `-bash
                `-pstree -u -a -A
```

The top-most `sshd -D` process is the actual server accepting new incoming connection requests. When a connection arrives, a child process is created processing this specific connection. This child process creates another child process, which executes using the permissions of the user that was authenticated. (The user then runs the shell `bash` and the shell command `pstree -u -a -A` to obtain the process tree.)



DNS Security Extensions (DNSSEC)

21 Pretty Good Privacy (PGP)

22 Transport Layer Security (TLS)

23 Secure Shell (SSH)

24 DNS Security Extensions (DNSSEC)

YouTube: DNSSec Explained

The DNS essentially implements a mapping of DNS names to typed resource records. The resource records of the same type linked to the same name form a resource record set.

To sign a DNS zone, it is necessary to first create an asymmetric public/private key pair. The public key is stored in a DNSKEY resource record. Using the private key, different resource record sets of a zone are signed.

- see Example 2.6 of Roland's thesis!

To obtain trust in the public key, a parent zone can use a DS resource record to reference a public key in a child zone.

The most common configuration for signed zones is to have two keys.

* The first key is called the Key Signing Key (KSK). This key is the secure entry point for the zone and is only used to generate a signature over the DNSKEY RRset.

* The second key is called the Zone Signing Key (ZSK). This key is used to generate the actual signatures over the RRsets in the zone.

Next to signing records in a zone, DNSSEC also generates cryptographically signed proofs of non-existence. These allow validators to verify that the name and record type in a query, for which they have received an NXDOMAIN response, do indeed not exist. However, different proposals have been made and work is still in progress to find a ”good” solution.

Example:

dig +dnssec ripe.net

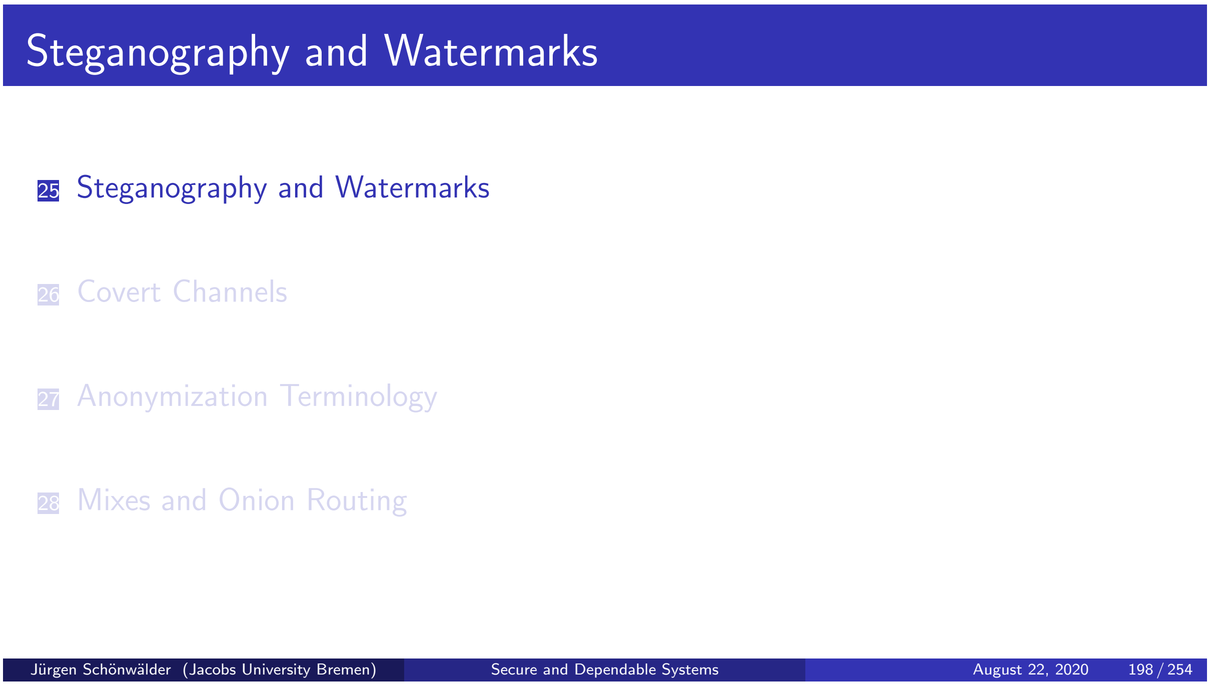
Part VI

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 in addition 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 [49].

We will first introduce techniques to hide data in other data (steganography) and ways 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). Afterwards, we look at basic principles of mixing networks and onion routing networks.



Steganography and Watermarks

25 Steganography and Watermarks

26 Covert Channels

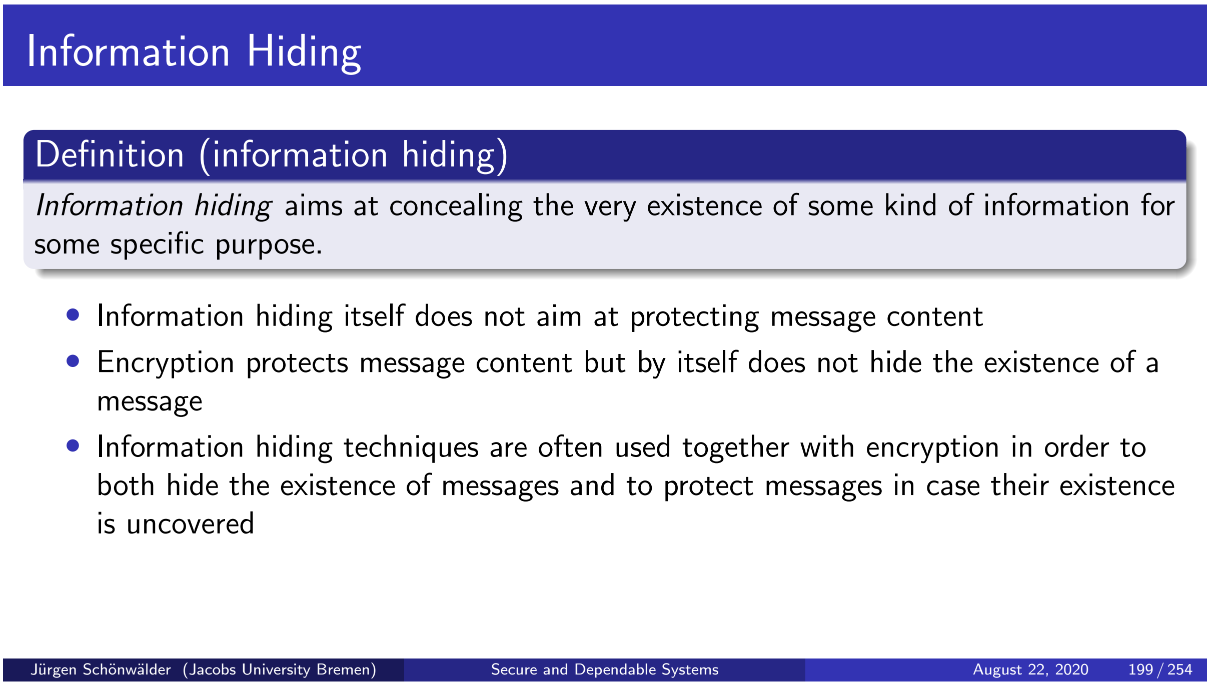
27 Anonymization Terminology

28 Mixes and Onion Routing

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 198 / 254

For a good introduction into stegangraphy, see the paper by Niels Provos and Peter Honeyman [39].

YouTube: Secrets Hidden in Images (Steganography) - Computerphile



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

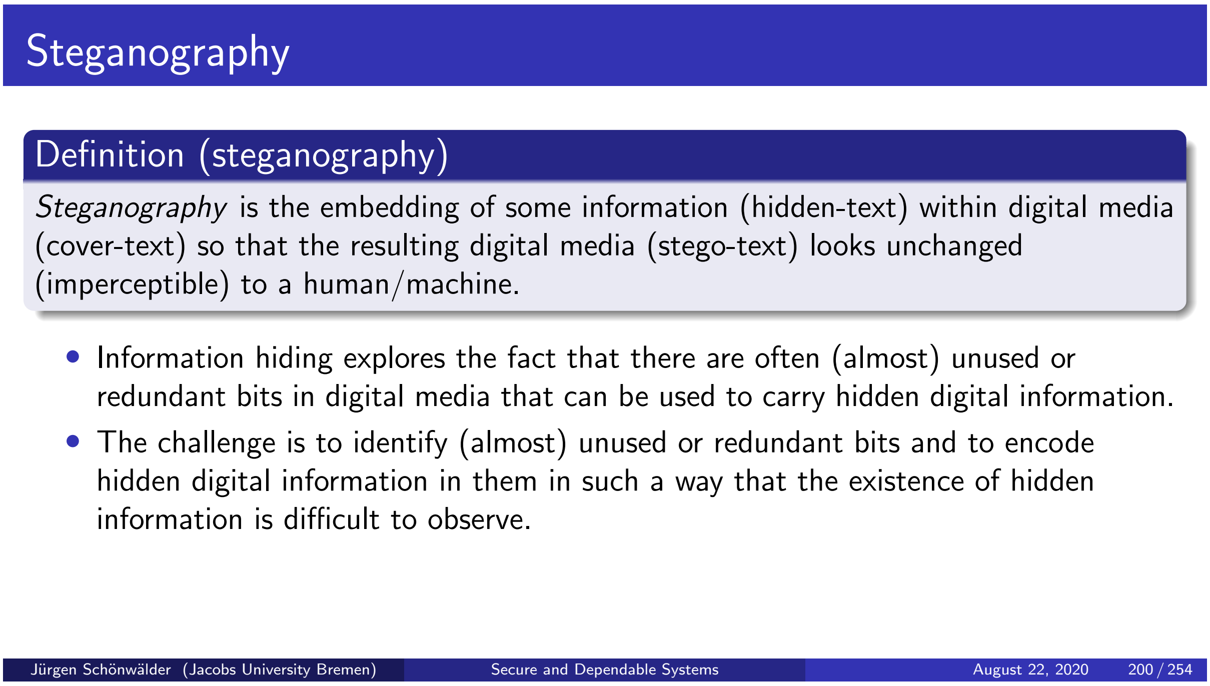
Some applications of information hiding:

- Improving confidentiality by hiding the very existence of messages
- Proving ownership of digital media by inserting hidden information into it (watermarking)
- Fingerprinting media for tracking purposes
- Hiding communication (covert channels)
- Identification of devices used to produce an artefact (e.g., printers embedding an identification code into printed documents)
- Hiding malware from being easily detected
- Enabling forensics, for example, by embedding identification codes into software
- Carrying data or programs through border control checks
- Storing sensitive information (e.g., passwords) in a way that can't be easily discovered.
- ...

The more you think of information hiding, the more possible applications you will likely discover. And at some point, you will be asking yourself what is stored in all the cat images and videos you can find on the Internet.

xkcd: Cat Proximity

xkcd: In Ur Reality



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.

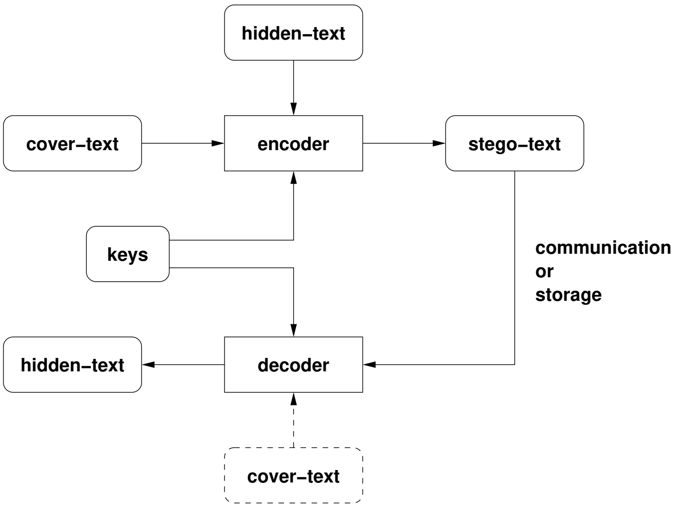
- Information hiding explores the fact that there are often (almost) unused or redundant bits in digital media that can be used to carry hidden digital information.
- The challenge is to identify (almost) unused or redundant bits and to encode hidden digital information in them in such a way that the existence of hidden information is difficult to observe.

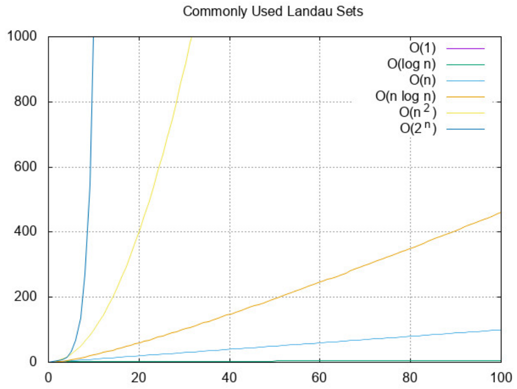
A very simple approach to embed hidden information into images is to take the color values (red, green, blue) of the pixels and to modify the least significant bits to encode hidden information. This is very straight-forward to implement and for humans the changes are almost impossible to spot (if you choose the pixels well). On the other hand, it is relatively easy to that an image might contain hidden information since the statistical properties of the pixels change.

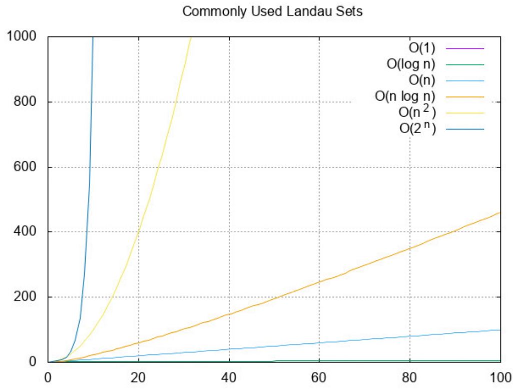
From an algorithmic point of view, this calls for smarter algorithms that can hide information without changing the the statistical properties of the media. And of course others will invent smarter detection algorithms.

For someone just interested in hiding information, there is another dimension since you can often choose the media in which to hide information: It is possible to write a script that embeds some given information to be hidden in many different media files and then selects the one where the change of the statistical properties of the media is smallest.

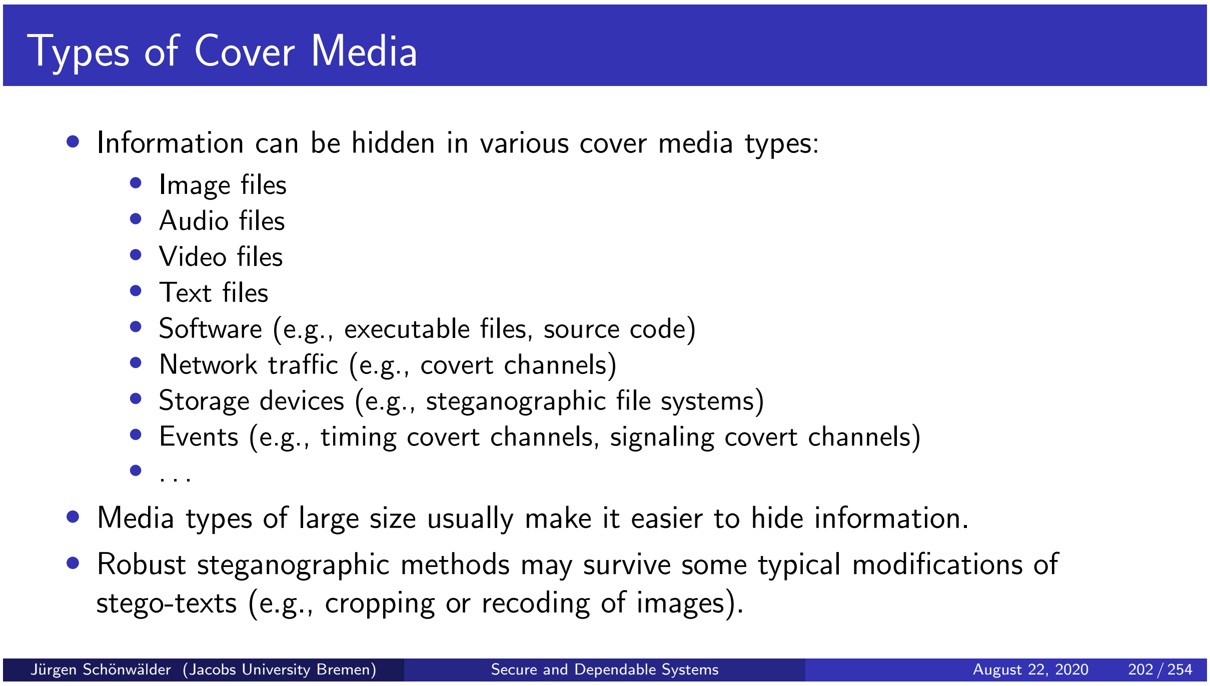
Steganography Workflow







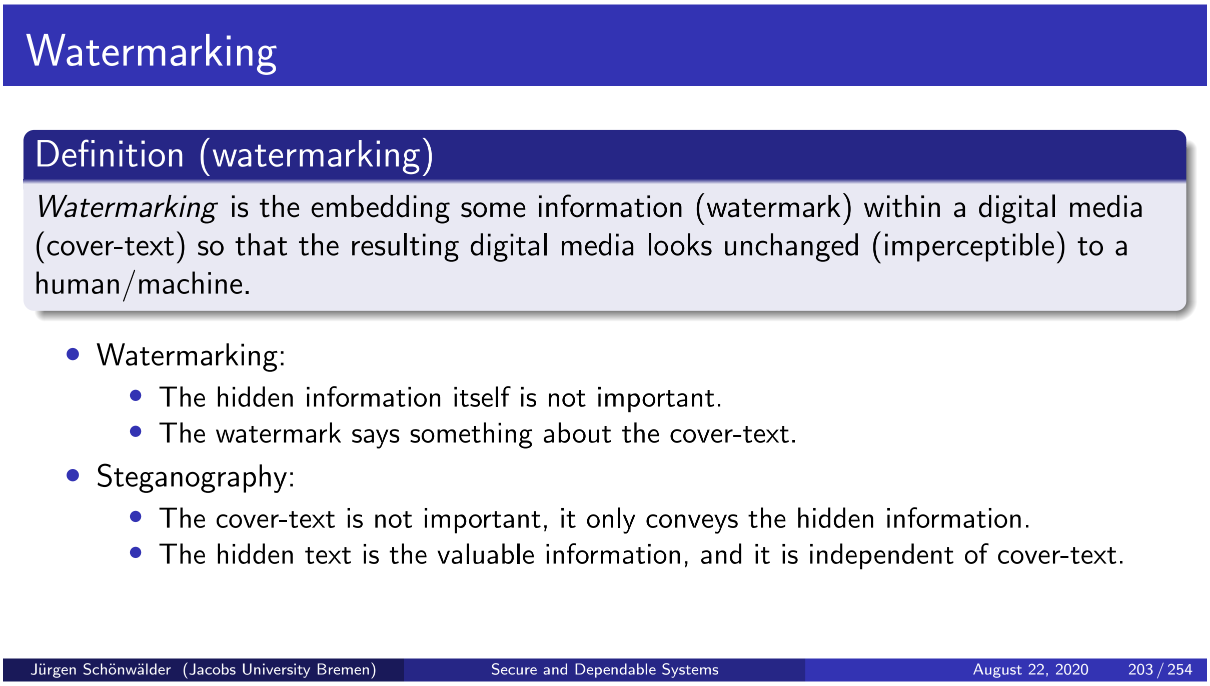
```
$ gnuplot landau.gp > landau.jpg
$ steghide embed -cf landau.jpg -ef landau.gp -sf landau.jpeg
$ steghide extract -sf landau.jpeg -xf -
set term jpeg
set title "Commonly Used Landau Sets"
set grid
set xrange [0:100]
set yrange [0:1000]

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).

Several steganographic file systems have been prototyped [4, 33, 7]. The basic idea is to fill unused blocks with random data and to store hidden files in these data blocks. Since the regular file system considers these blocks as free space, these blocks may be allocated and overwritten. Hence, a steganographic file system has to keep hidden information in a sufficiently large number of (redundant) seemingly unused data blocks.



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 typically designed to produce robust watermarks (watermarks that survive transformations applied to the cover text) and to create watermarks that are difficult to detect and remove.

One specific application of watermarking is the detection of modifications of digital media. Image processing tools, for example, 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

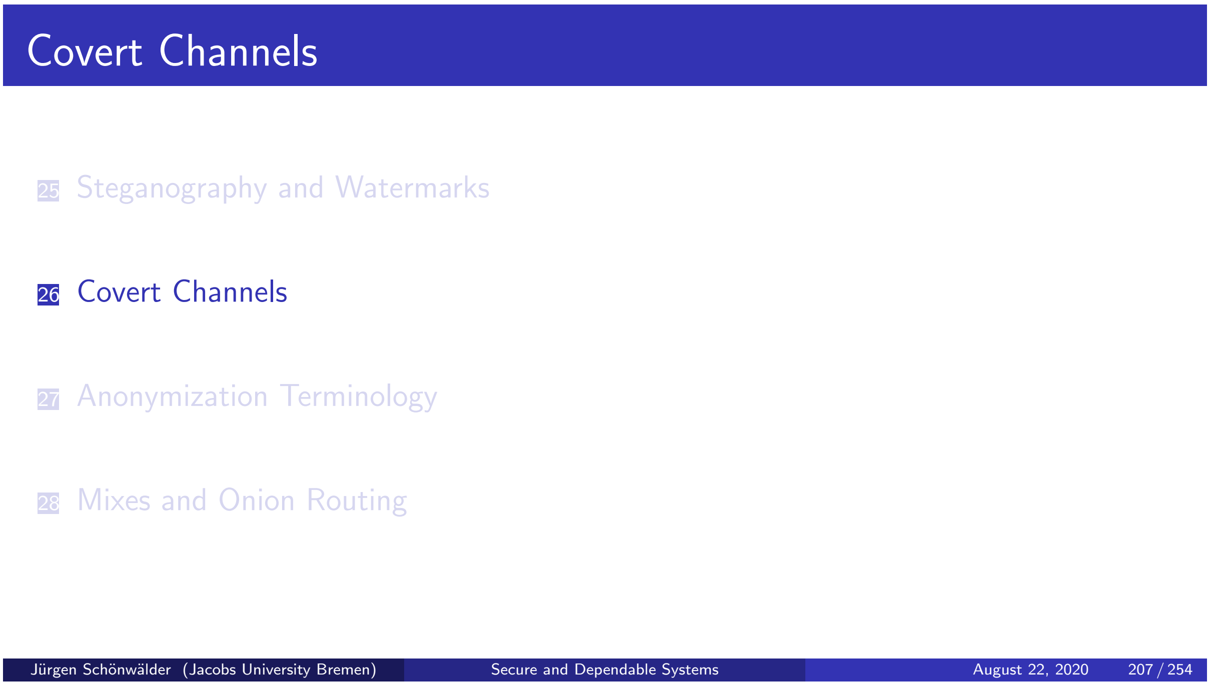
- fragile vs. robust
  - Fragile: Modifiations of stego-text likly destroys hidden text.
  - Robust: Hidden text is likely to survive modifications of the stego-text.
- blind vs. semi-blind vs. non-blind
  - Blind requires the original cover-text for detection / extraction.
  - Semi-blind needs some 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.

Example: 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.

Example: DCT-based Image Steganography

- Idea:
  - Some image formats (e.g., 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

25 Steganography and Watermarks

26 Covert Channels

27 Anonymization Terminology

28 Mixes and Onion Routing

For an overview of covert channel, see the survery paper by Steffen Wendzel et al. [50].

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 size of protocol data units
  - 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.

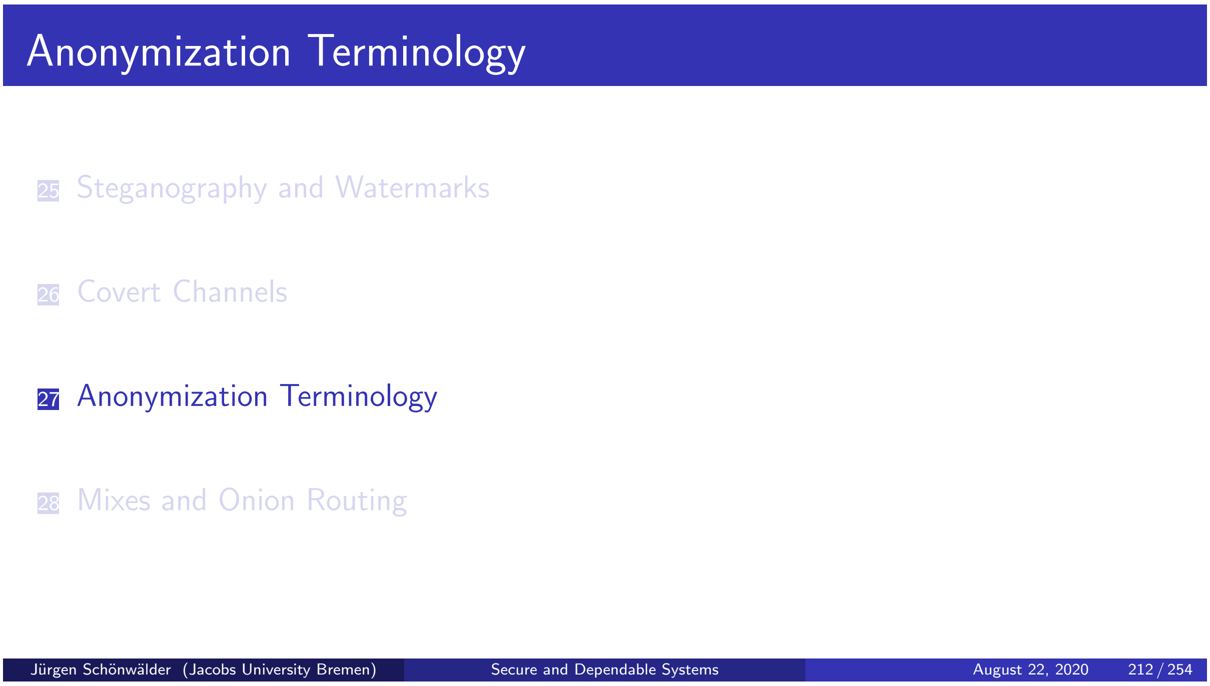
P8 Inter-arrival Time Pattern
The covert channel alters timing intervals between protocol messages (inter-arrival times) to encode hidden data.

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

25 Steganography and Watermarks

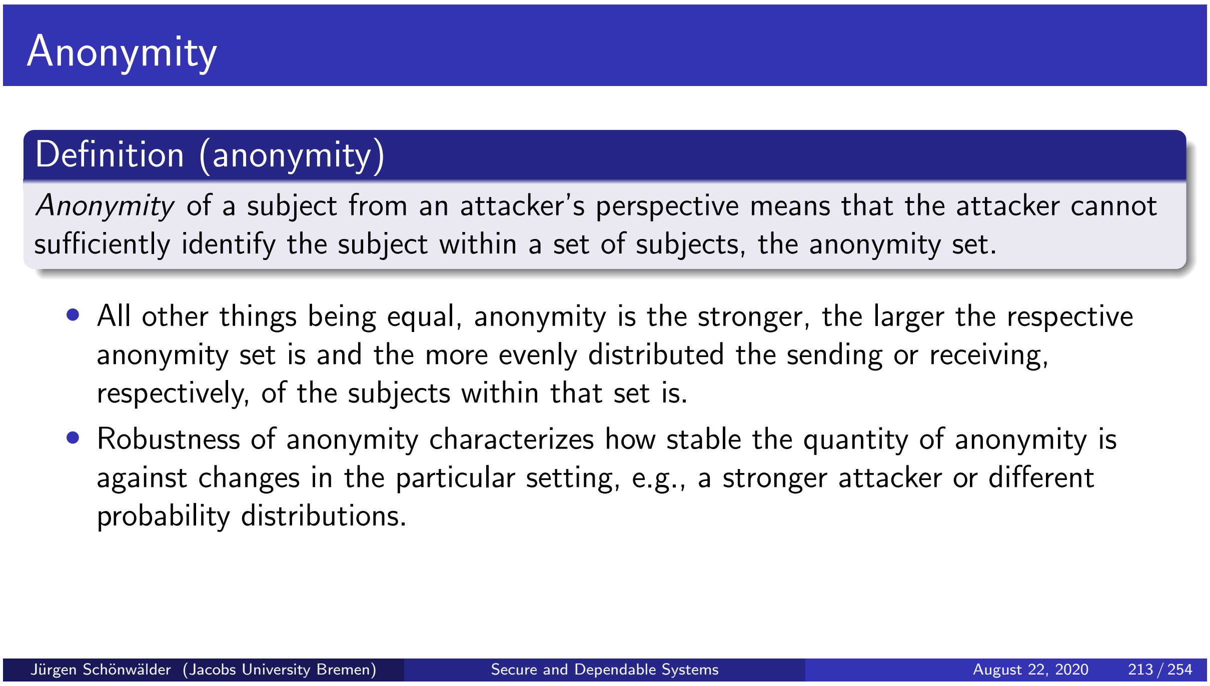
26 Covert Channels

27 Anonymization Terminology

28 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



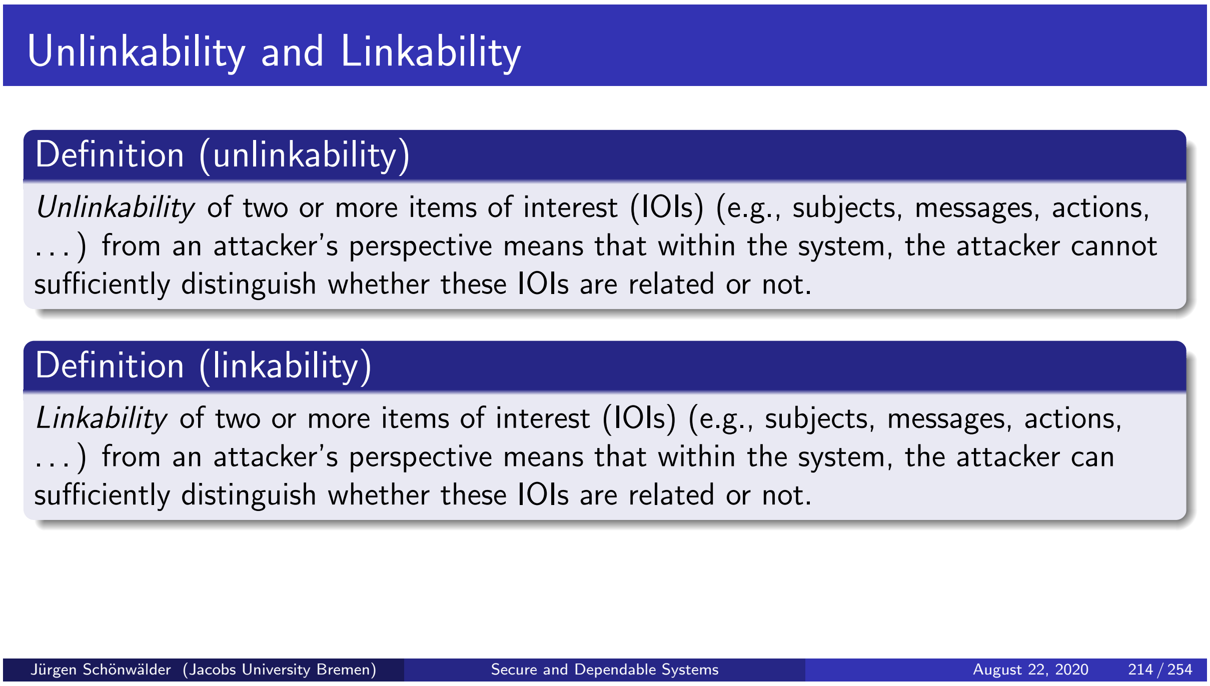
Anonymity

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.

It is important to consider the anonymity set when we talk about anonymity. Obviously, being anonymous among 20 students feels weaker than being anonymous among 200 students or 2000 students. Unfortunately, in many real life situations, the anonymity set is not really known.



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.

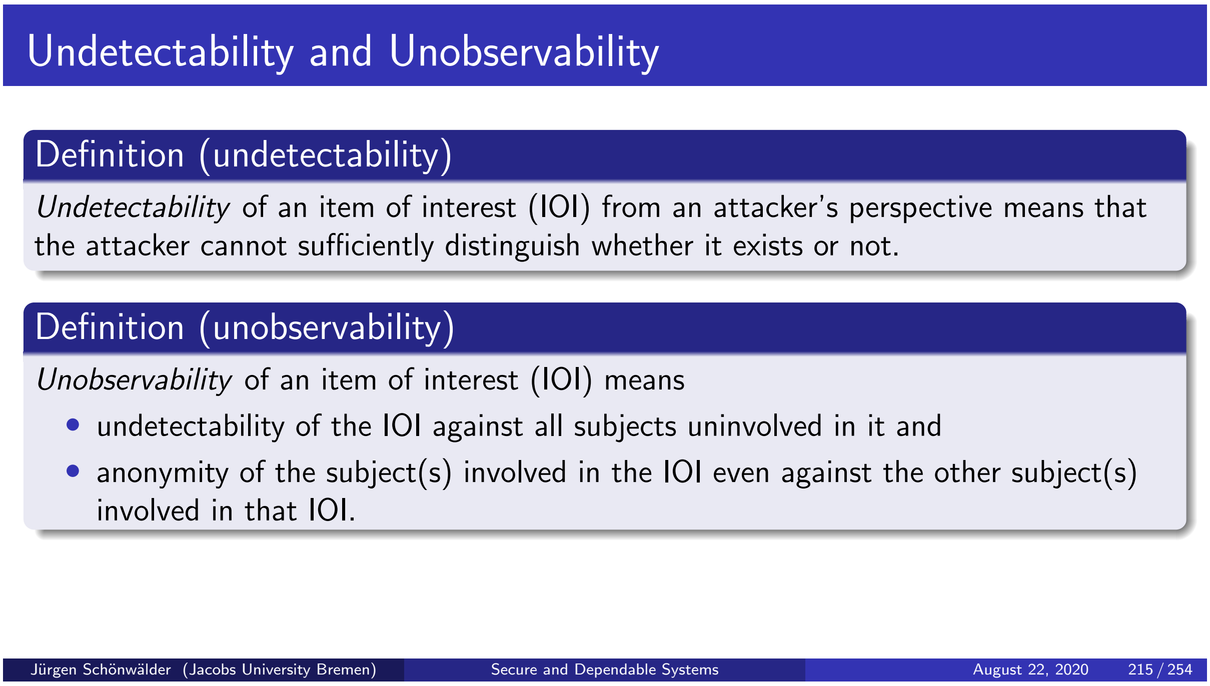
Definition (linkability)

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.

When we talk about about sender anonymity, the anonymity set is the set of all senders, the sender anonymity set. Similiarly, when we talk about recipient anonymity, the anonymity set is the set of all recipients, i.e., the recipient anonymity set.



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.

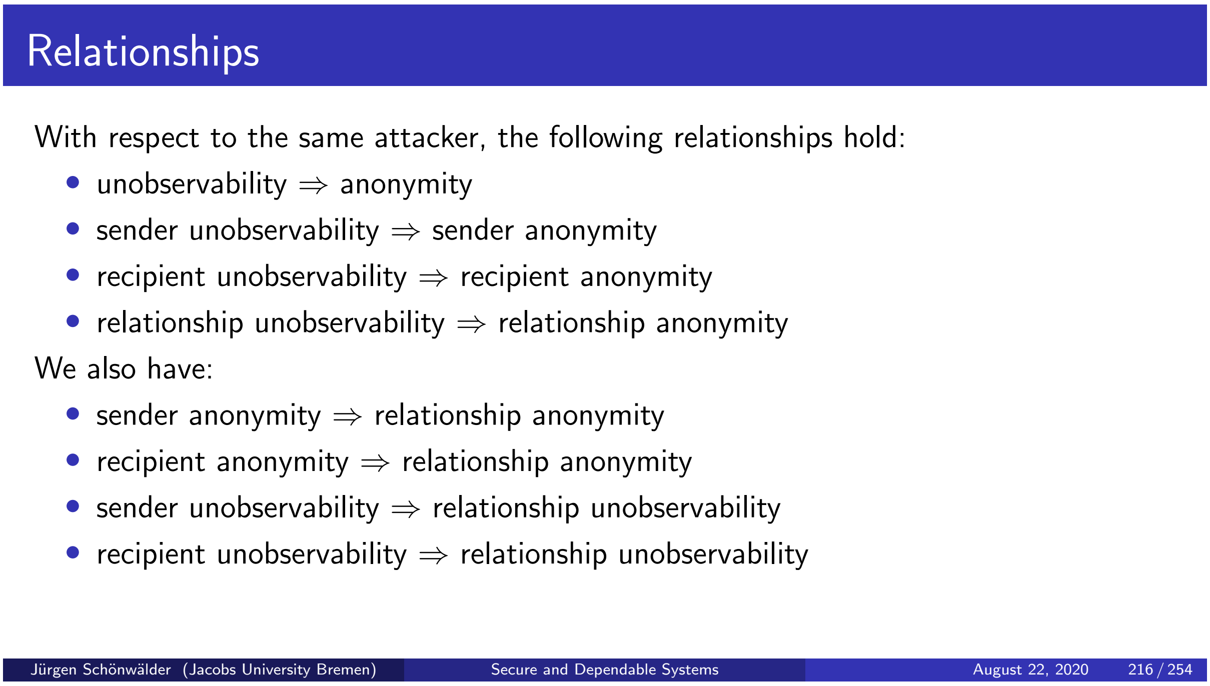
Definition (unobservability)

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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 215 / 254

- *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

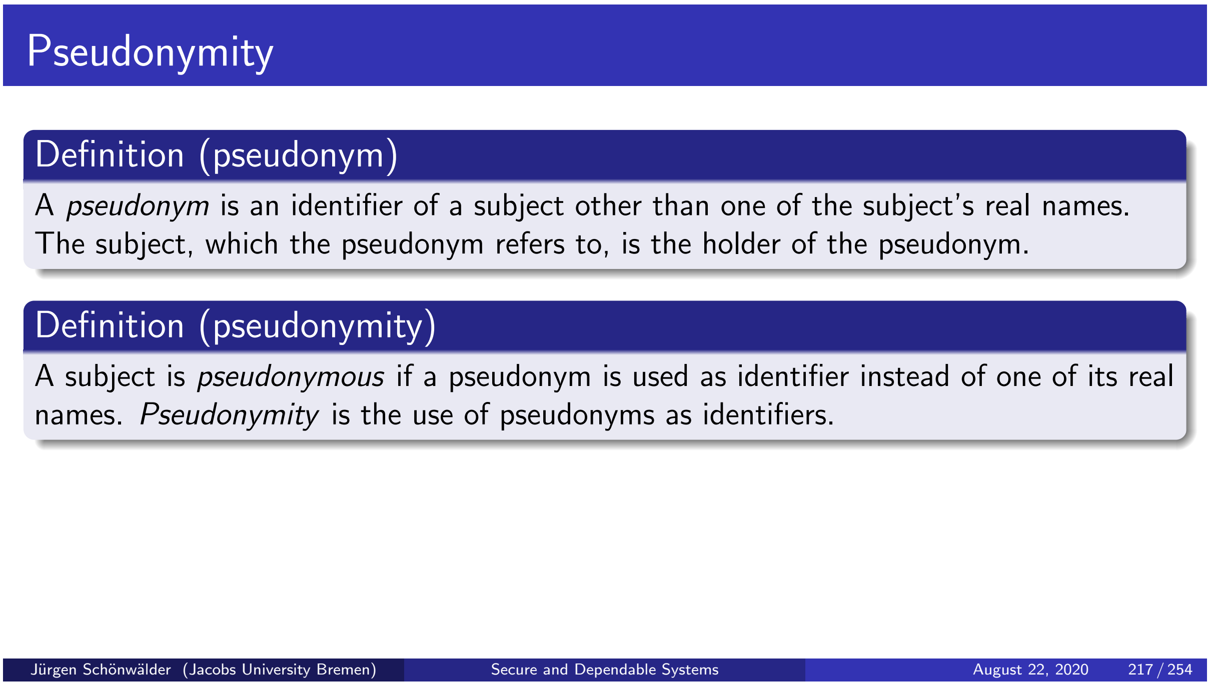
With respect to the same attacker, the following relationships hold:

- 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 items of interest (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

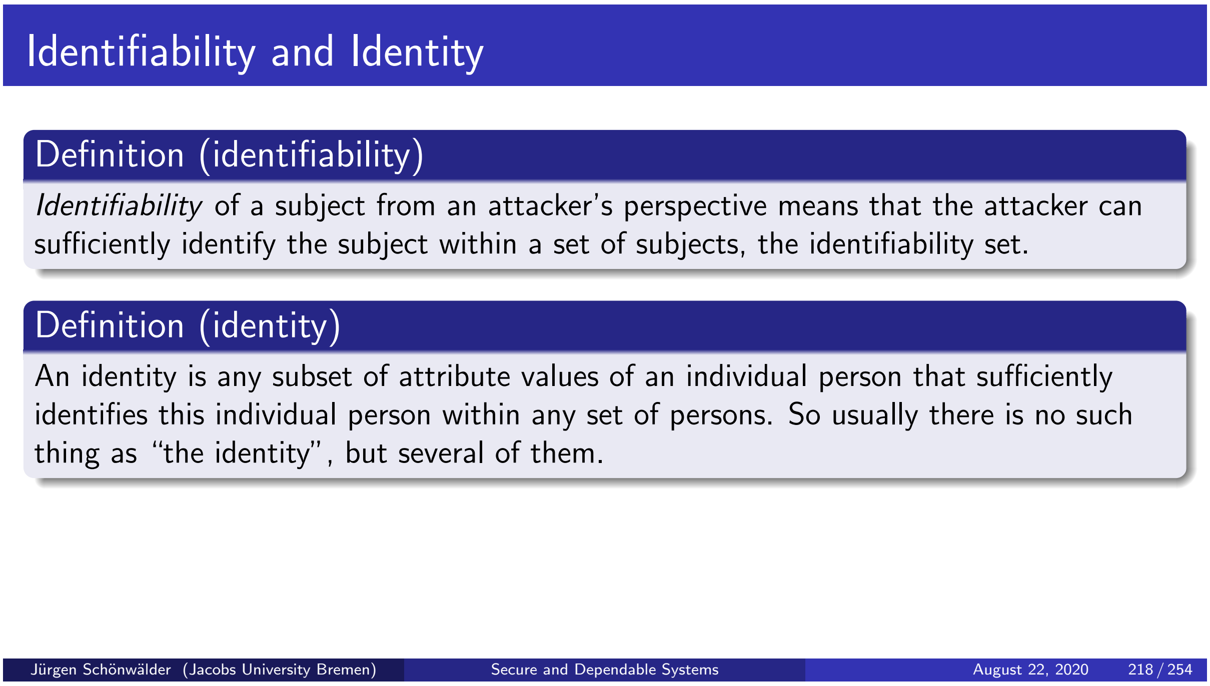
Definition (pseudonym)

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*.
- The relation between a pseudonym and the related subject can be thought of as “a subject holds a pseudonym” and in general a subject hold one or multiple pseudonyms.



Identifiability and Identity

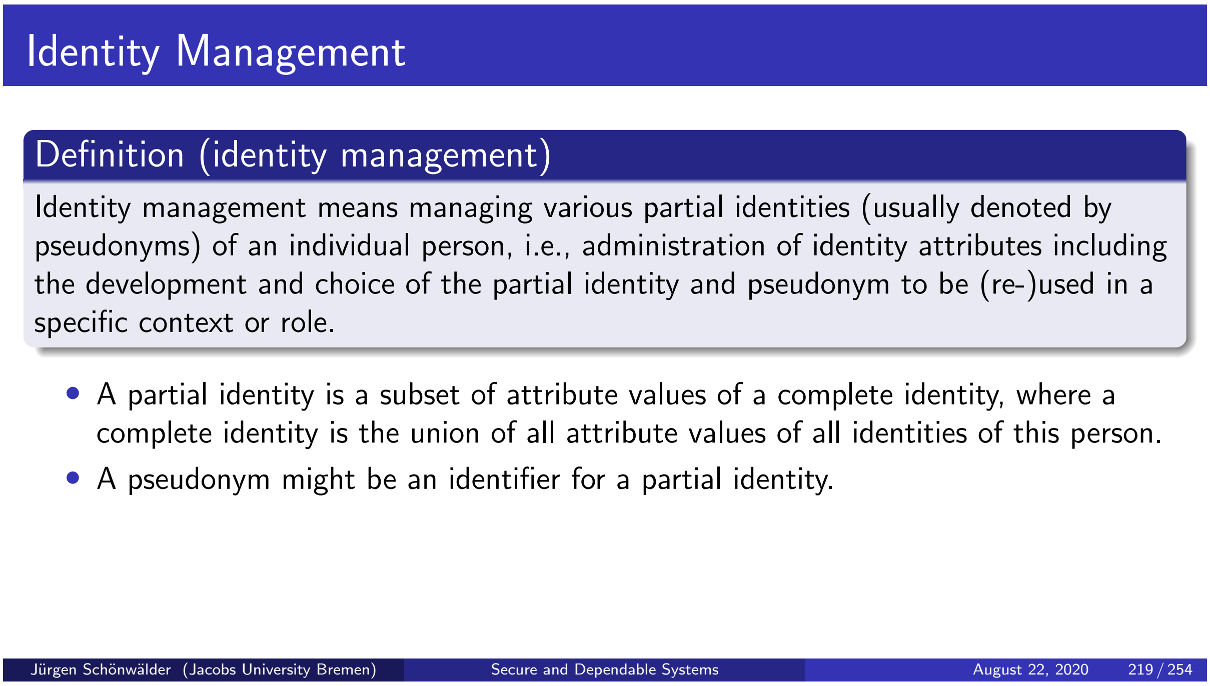
Definition (identifiability)

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 that 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 items of interest (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.

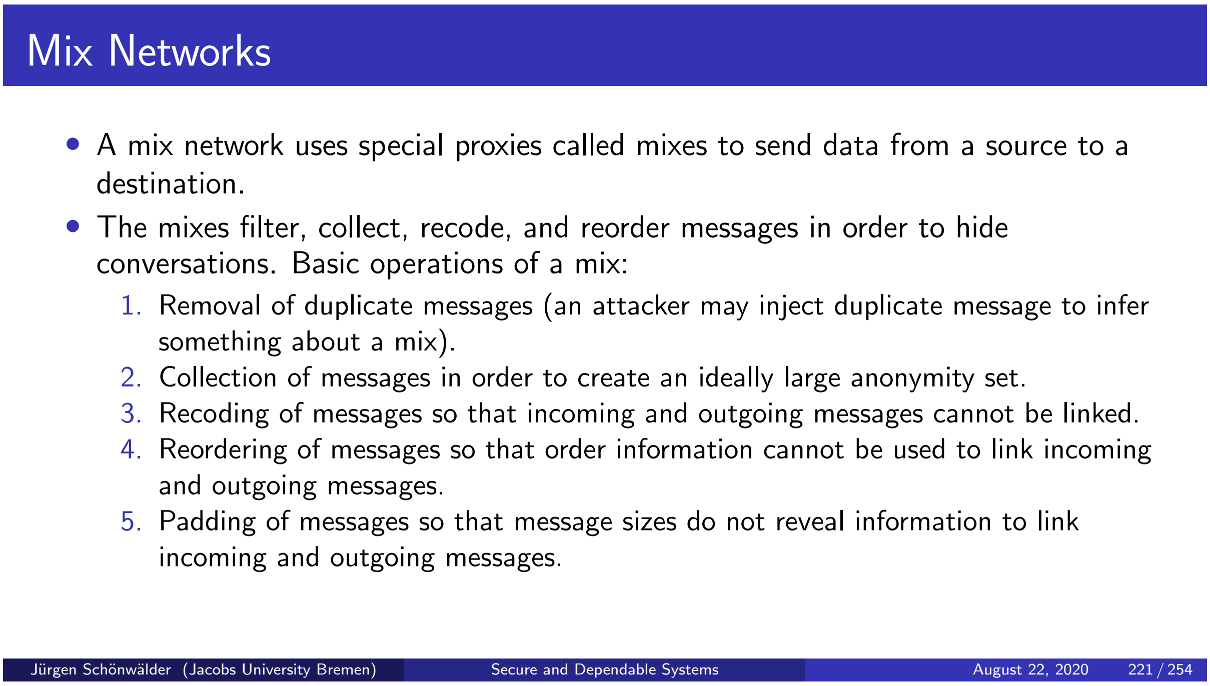
Mixes and Onion Routing

25 Steganography and Watermarks

26 Covert Channels

27 Anonymization Terminology

28 Mixes and Onion Routing



Mix Networks

- A mix network uses special proxies called mixes to send data 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 [13]. Mix networks get their security from the mixing done by their component mixes, and may or may not use route unpredictability to enhance security. We use the following 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: $(A \to M_1 \to M_2 \to B)$

$$\begin{aligned} A \to M_1 &: \{N_1, M_2, \{N_2, B, \{m\}_{K_B}\}_{K_{M_2}}\}_{K_{M_1}} && M_1 \text{ extracts } M_2 \\ M_1 \to M_2 &: \{N_2, B, \{m\}_{K_B}\}_{K_{M_2}} && M_2 \text{ extracts } B \\ M_2 \to B &: \{m\}_{K_B} && B \text{ extracts the message } m \end{aligned}$$

- 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:

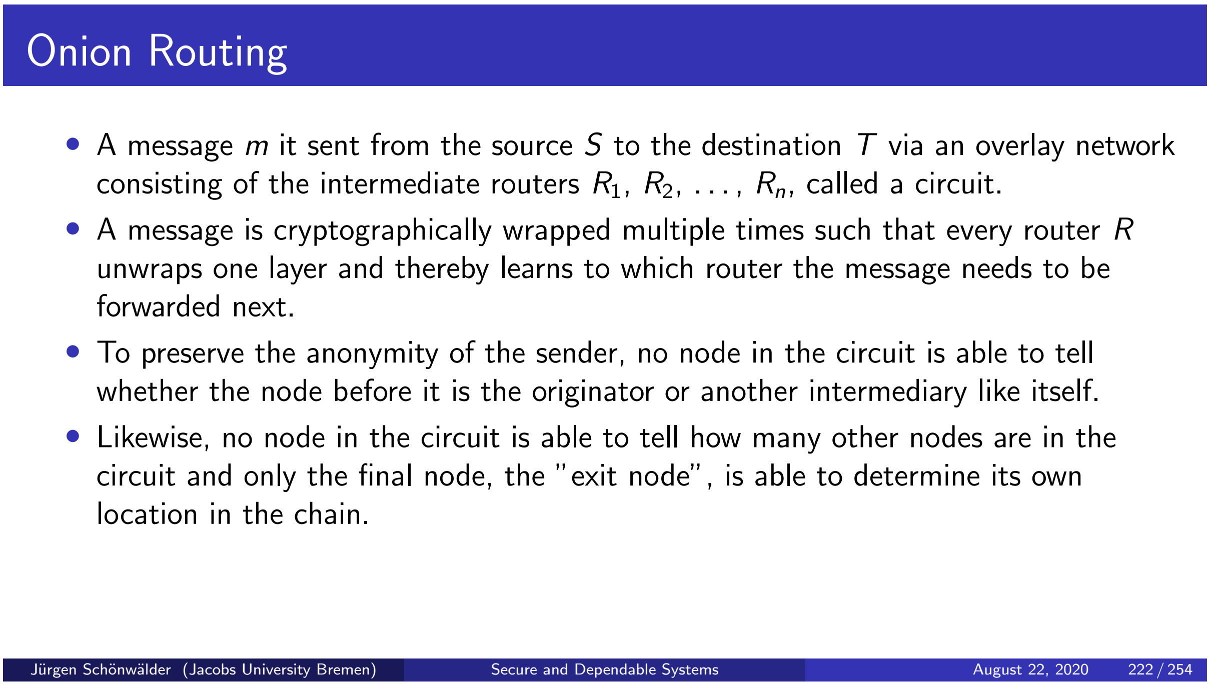
$$R = \{k_0, M_1, \{k_1, M_2, \{k_2, B\}_{K_{M_2}}\}_{K_{M_1}}\}_{K_A}$$

  The return address is sent to A as described above:

$$\begin{aligned} B \to M_2 &: \{N_2, M_1, \{N_1, A, \{R\}_{K_A}\}_{K_{M_1}}\}_{K_{M_2}} && M_2 \text{ extracts } M_1 \\ M_2 \to M_1 &: \{N_1, A, \{R\}_{K_A}\}_{K_{M_1}} && M_1 \text{ extracts } A \\ M_1 \to A &: \{R\}_{K_A} && A \text{ extracts the return address } R \end{aligned}$$

  The sender A extracts k_0 from R and sends the following to M_1:

$$\begin{aligned} A \to M_1 &: \{m\}_{k_0}, \{k_1, M_2, \{k_2, B\}_{K_{M_2}}\}_{K_{M_1}} && M_1 \text{ extracts } k_1 \text{ and } M_2 \\ M_1 \to M_2 &: \{\{m\}_{k_0}\}_{k_1}, \{k_2, B\}_{K_{M_2}} && M_2 \text{ extracts } k_2 \text{ and } B \\ M_2 \to B &: \{\{\{m\}_{k_0}\}_{k_1}\}_{k_2} && B \text{ extracts the message } m \end{aligned}$$



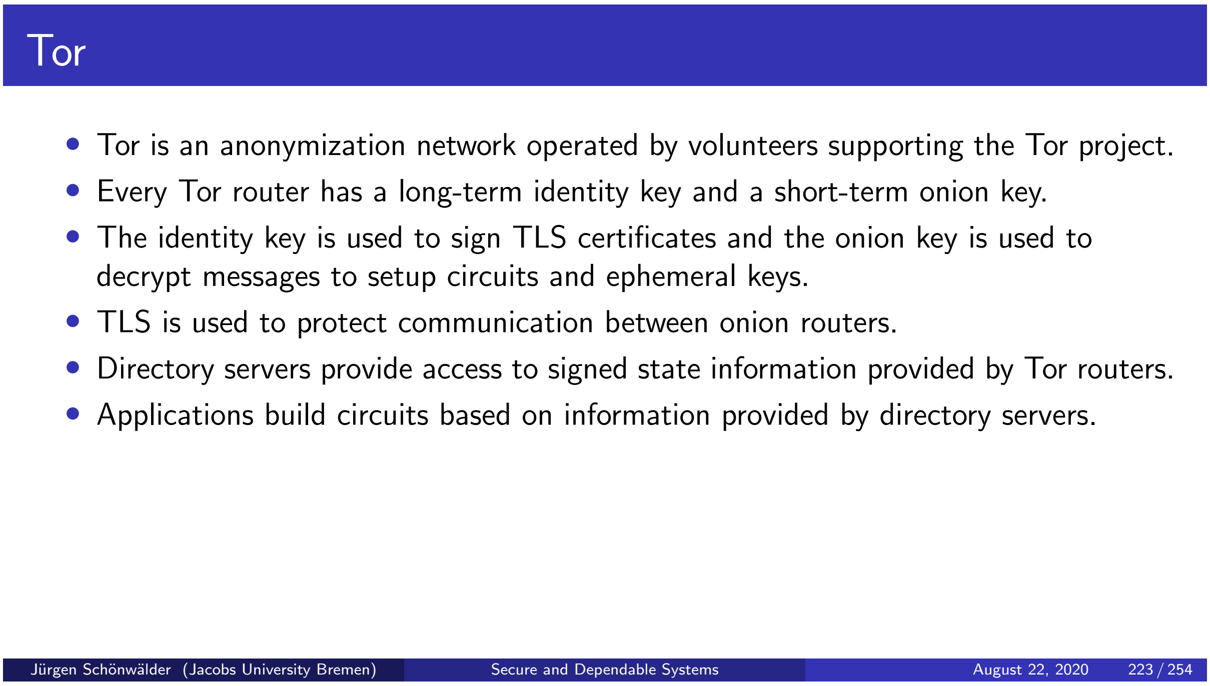
Onion Routing

- 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, $\ldots$, 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. Onion routing systems can provide access to real-time services. The security of onion routing systems rests on the assumption that not all routers in an onion routing system can be controlled by an adversary.

YouTube: Onion Routing - Computerphile

YouTube: Mix Networks (Mixnets) by Jaime Lee Pabilona



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.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 223 / 254

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 [18].

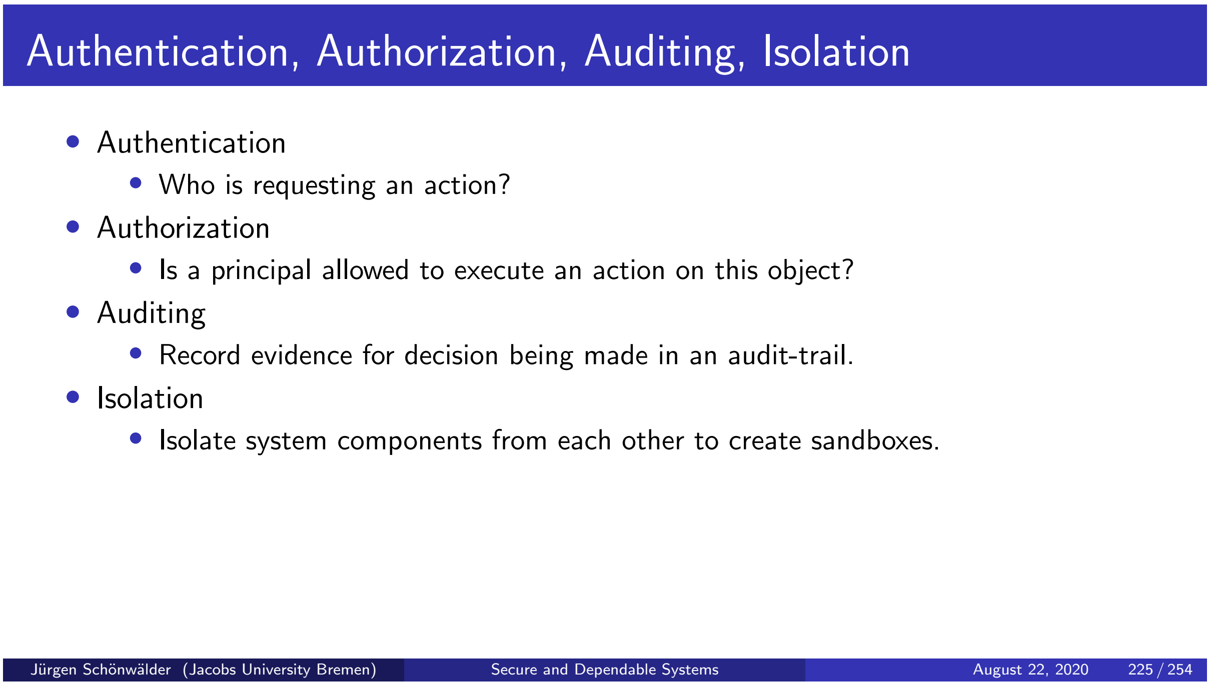
Tor aims at protecting the traffic in transit, it is of limited help if application protocols running over Tor circuits leak information that allows to link traffic to identities. Since Tor does aim at supporting interactive applications, it is in general subject to traffic analysis attacks and in particular 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

System Security

It is now time to take a look at computer security from a systems perspective. We will focus on authentication, authorization, and auditing and on isolation. Before we look at how operating systems implement these functions, we take a look at trusted computing architectures, that introduce special hardware features that can be used to bootstrap trust into other software components.



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.

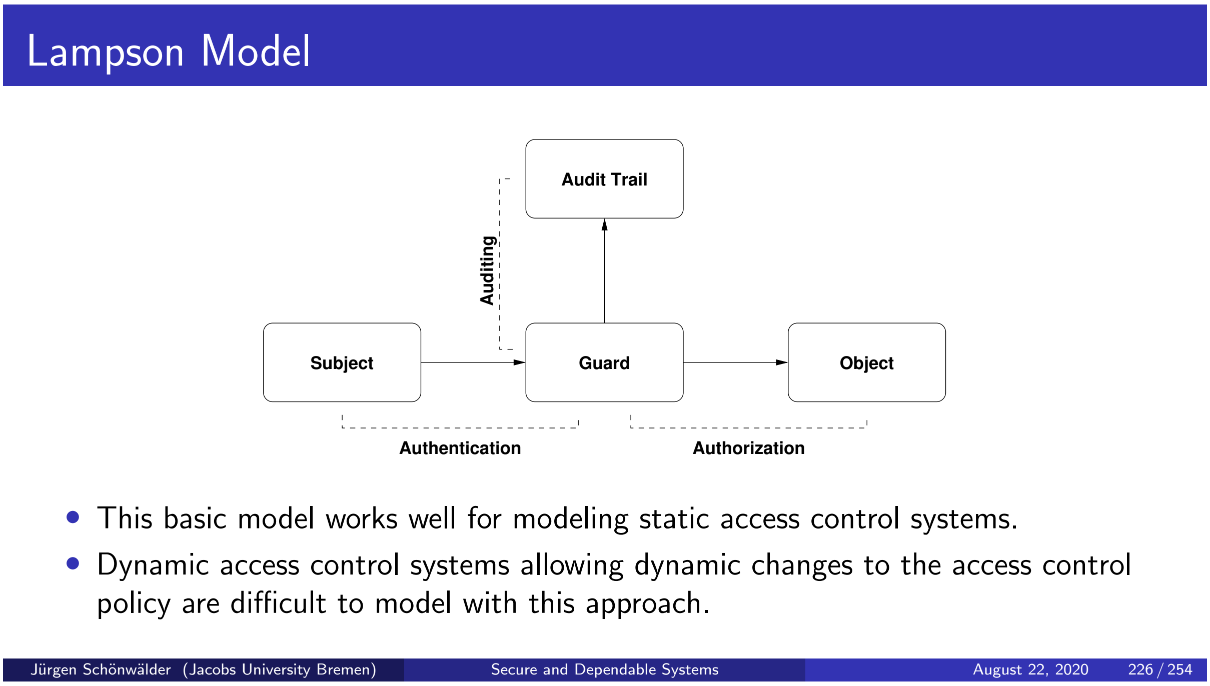
Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 225 / 254

Basic authentication at the 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.

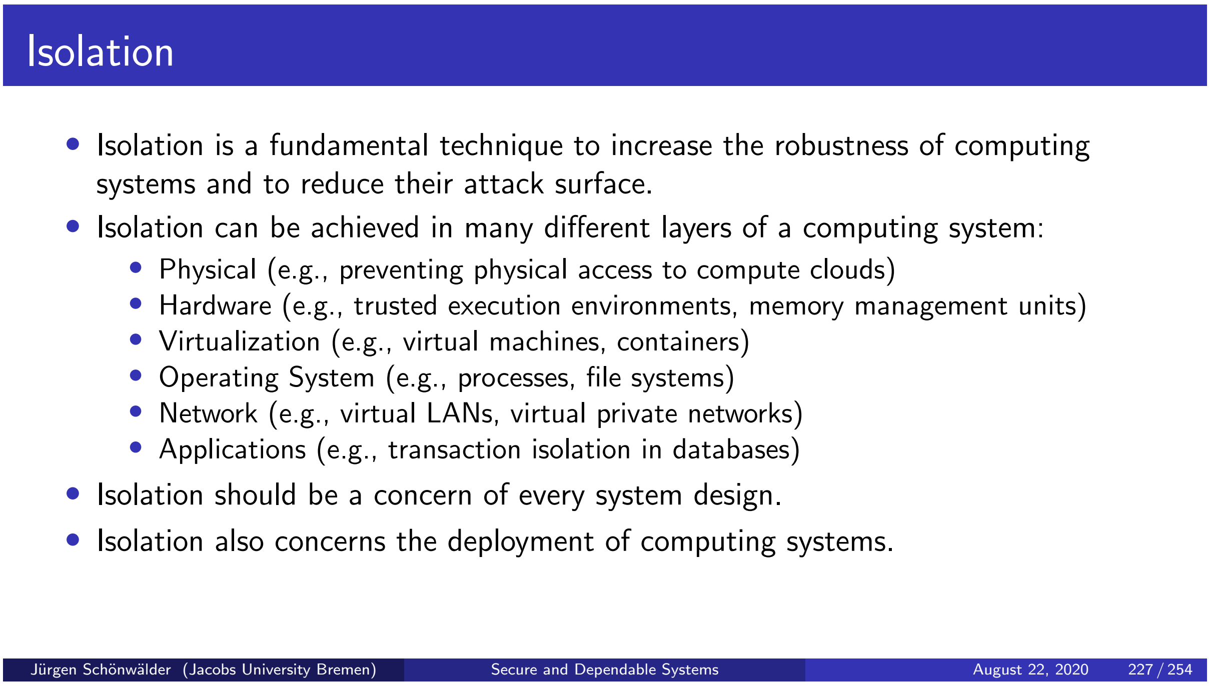


Lampson Model

Audit Trail
Auditing
Subject
Guard
Object
Authentication
Authorization

- 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 [30] uses a slightly different terminology.

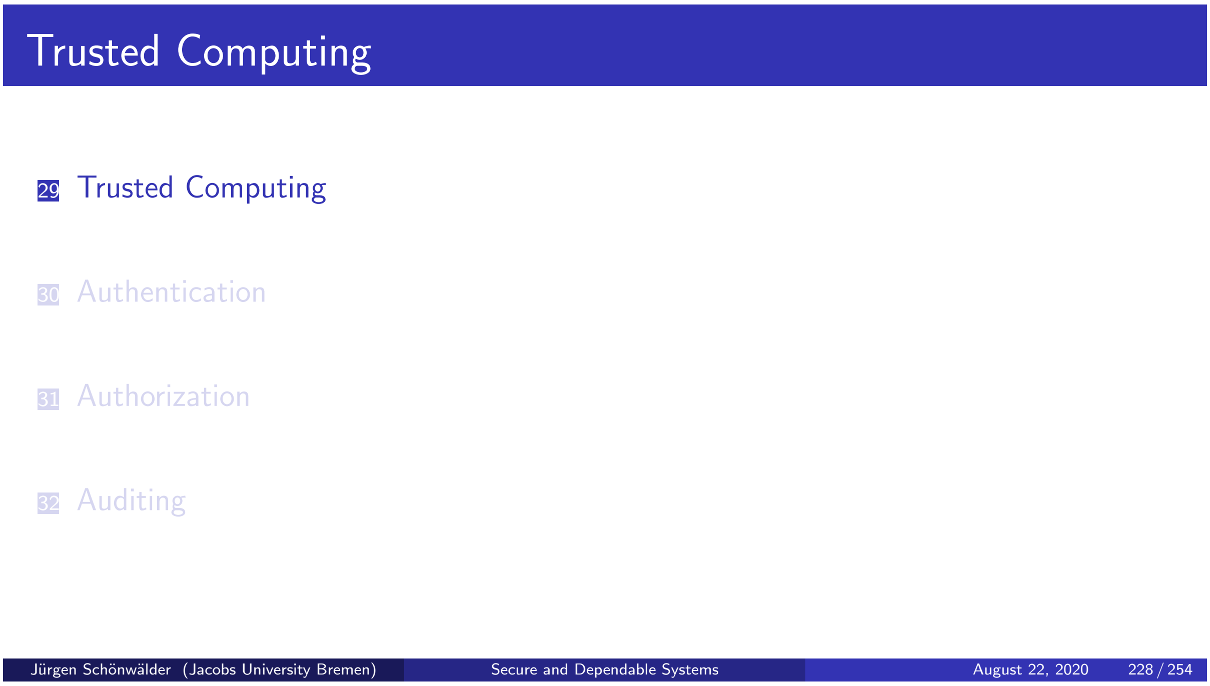


Isolation

- Isolation is a fundamental technique to increase the robustness of computing systems and to reduce their attack surface.
- Isolation can be achieved in many different layers of a computing system:
  - Physical (e.g., preventing physical access to compute clouds)
  - Hardware (e.g., trusted execution environments, memory management units)
  - Virtualization (e.g., virtual machines, containers)
  - Operating System (e.g., processes, file systems)
  - Network (e.g., virtual LANs, virtual private networks)
  - Applications (e.g., transaction isolation in databases)
- Isolation should be a concern of every system design.
- Isolation also concerns the deployment of computing systems.

It is important to consider isolation not only during the design of software but also when computing systems and software gets deployed. Systems that have designed with isolation in mind tend to resist attacks much better while attacks on deployed systems that lack a proper isolation can of course much more significant damage. The downside is that maintaining proper isolation often makes operational processes more complicated.

A paper discussing several system-level security isolation techniques is [47].



Trusted Computing

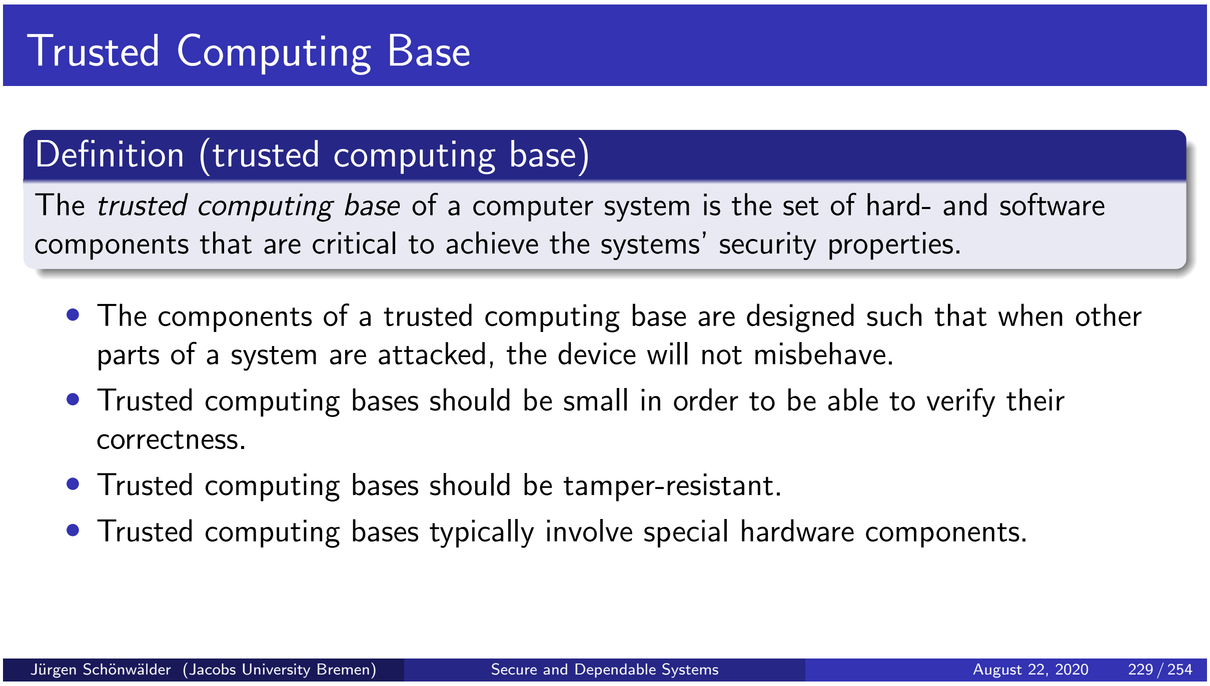
29 Trusted Computing

30 Authentication

31 Authorization

32 Auditing

A good overview of hardware-based trusted computing architectures can be found in [32].



Trusted Computing Base

Definition (trusted computing base)

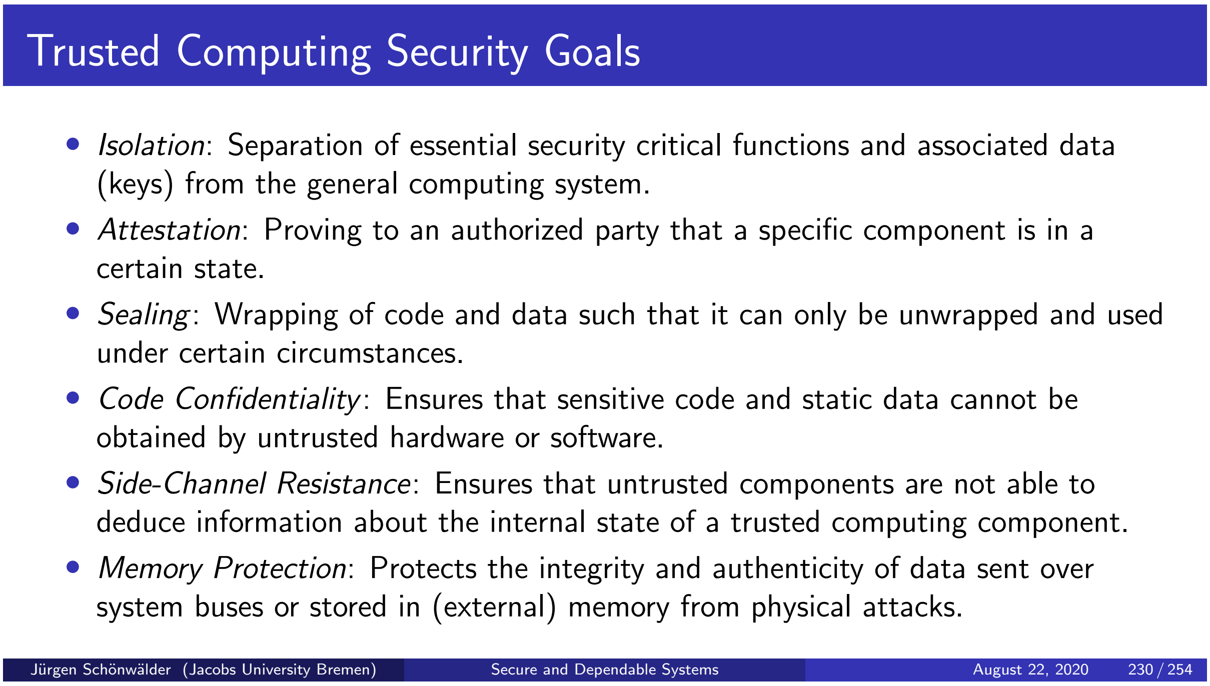
The *trusted computing base* of a computer system is the set of hard- and software components that are critical to achieve the systems' security properties.

- The components of a trusted computing base are designed such that when other parts of a system are attacked, the device will not misbehave.
- Trusted computing bases should be small in order to be able to verify their correctness.
- Trusted computing bases should be tamper-resistant.
- Trusted computing bases typically involve special hardware components.

The general idea behind trusted computing is to design hardware and software components that can be trusted. Since it is hard to design software that can be trusted, a trusted computing base typically includes hardware components that are assumed to be tamper-resistant. These trusted hardware components can then be used to to bootstrap trust into other software components, for example, an operating system that has been loaded using a secure boot process involving trusted hardware components.

Measurements are used to assess the authenticity of software components and data (e.g., by calculating a cryptographic hash over code and data). The measurements can be reported in order to attest the component's state to other systems. This process is called *attestation*. For example, an attestation may prove that a proper operating system kernel has been loaded into a computer created by a specific manufacturer.

Note that trusted computing is in particular of high relevance for the fast growing number of mobile and embedded devices. A modern car, for example, consists of many small embedded computer systems and there is certain interest to verify that the devices implementing critical functions of a car have not been tampered with.



Trusted Computing Security Goals

- *Isolation*: Separation of essential security critical functions and associated data (keys) from the general computing system.
- *Attestation*: Proving to an authorized party that a specific component is in a certain state.
- *Sealing*: Wrapping of code and data such that it can only be unwrapped and used under certain circumstances.
- *Code Confidentiality*: Ensures that sensitive code and static data cannot be obtained by untrusted hardware or software.
- *Side-Channel Resistance*: Ensures that untrusted components are not able to deduce information about the internal state of a trusted computing component.
- *Memory Protection*: Protects the integrity and authenticity of data sent over system buses or stored in (external) memory from physical attacks.

Isolation is the key motivation for defining trusted computing bases. In order to bootstrap trust into a system, a system needs to be able to hold keys in tamper-resistant memory and it must be able to perform cryptographic operations in such a way that keys never leave the isolated environment. As a consequence, the first candidates of functions to place into trusted hardware are cryptographic algorithms and key generation and storage. But once more flexibility is desired, it makes sense to add more functionality to the isolated environment and ultimately you will make the isolated environments programmable (which then eventually may lead to a recursion).

Attestation is often needed to verify that a system can be trusted. Attestation may be a local or remote process. For example, a car manufacturer may decide to install firmware updates only on devices that have not been tampered with. Hence, the software update process may request a remote attestation that the car component is in a proper state.

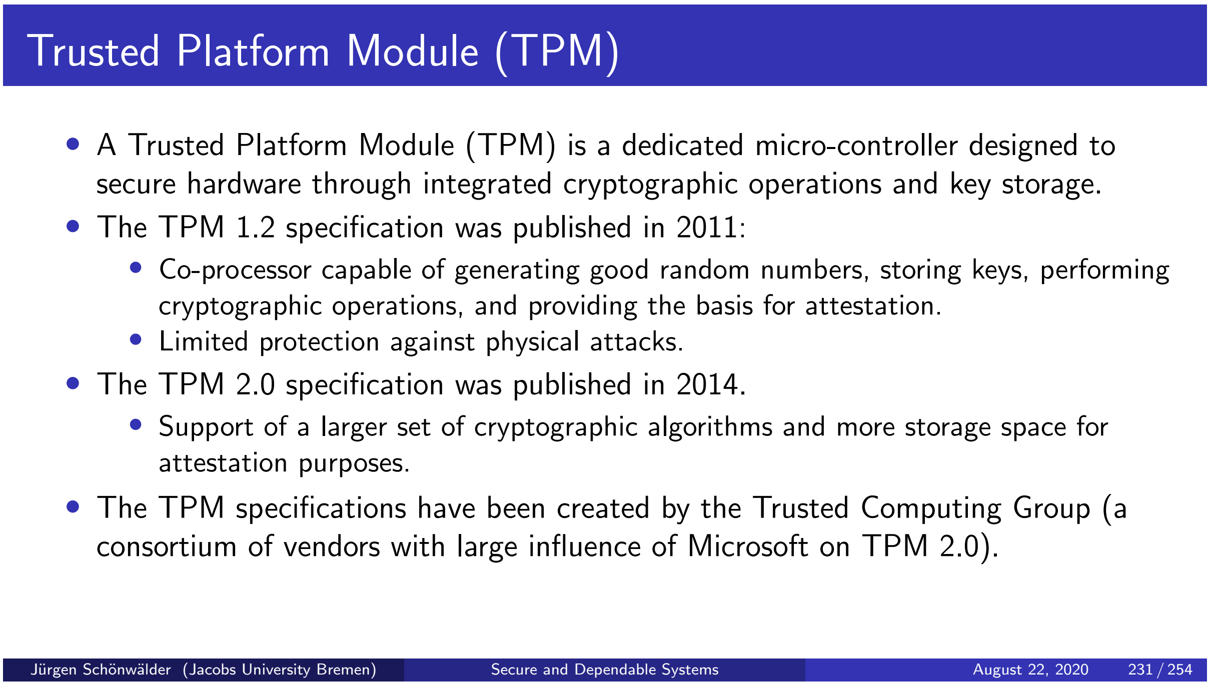
Sealing code and data can for example be used to bind it to a specific device, a certain configuration of a device, the state of a software module or a combination of these.

Code confidentiality may be used to protect intellectual property. Code confidentiality may be achieved by combining isolation, encryption, and sealing.

Side-channel resistance is an important property. The attacker model has a big influence on the costs for achieving side-channel resistance and hence this must be well defined in order to know what side-channel resistance means. For example, an attacker who has physical access to the hardware can launch attacks by injecting faults of measuring power consumption to reveal information about the internal state of a trusted computing component. An attacker who has only access to the untrusted computing components may reveal information via a timing side-channel attack on shared caches.

Memory protection has to consider passive attacks (e.g., bus snooping) and active attacks (e.g., data or fault injection). The means are to encrypt data, to calculated integrity checksums, and to prevent replay attacks. Of course, this is challenging to do at typical bus speeds.

Most trusted computing systems only support some of these security goals. The reason is simply that supporting all of them increases complexity, which defeats the purpose of keeping the trusted computing base small.



Trusted Platform Module (TPM)

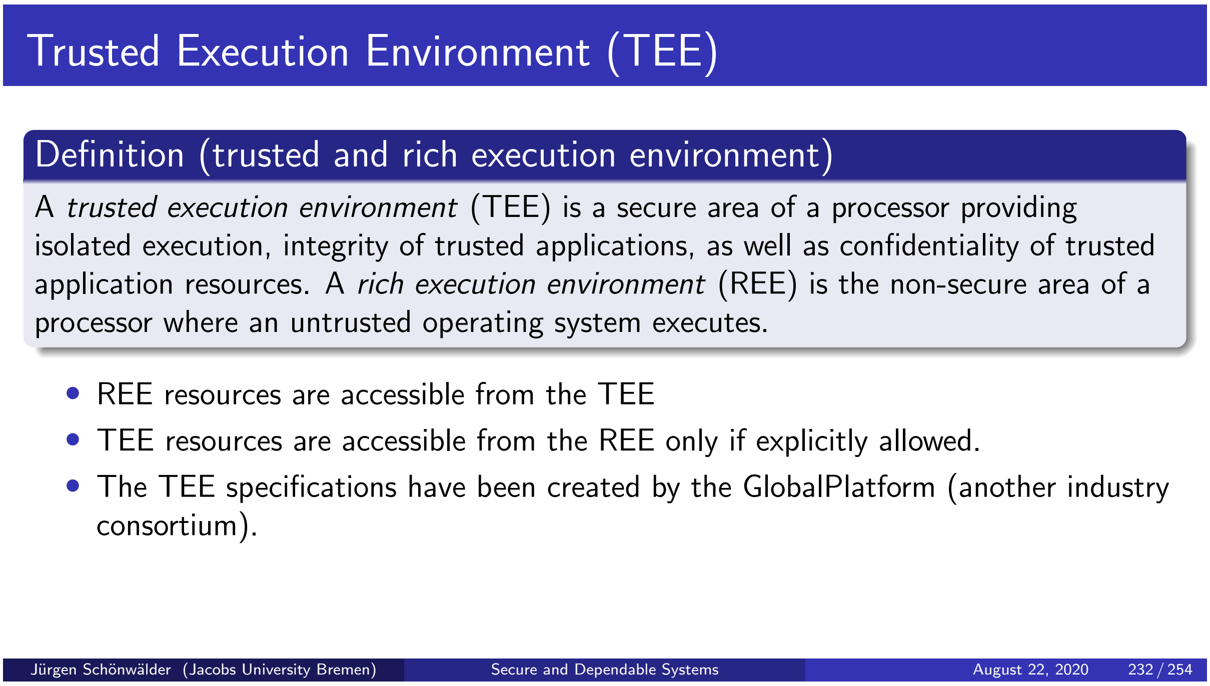
- A Trusted Platform Module (TPM) is a dedicated micro-controller designed to secure hardware through integrated cryptographic operations and key storage.
- The TPM 1.2 specification was published in 2011:
  - Co-processor capable of generating good random numbers, storing keys, performing cryptographic operations, and providing the basis for attestation.
  - Limited protection against physical attacks.
- The TPM 2.0 specification was published in 2014.
  - Support of a larger set of cryptographic algorithms and more storage space for attestation purposes.
- The TPM specifications have been created by the Trusted Computing Group (a consortium of vendors with large influence of Microsoft on TPM 2.0).

The TPM specification version 1.2 requires that TPMs support a random number generator (RNG), an RSA implementation of at least 2048 bit keys, and the SHA-1 cryptographic hash function. At manufacturing time, a so-called Endorsement Key (EK) is generated and burned into the TPM. This key identifies a particular TPM chip and implicitly that hardware that makes use of this chip. Additional keys such as Attestation Identity Keys (AIKs) can be generated and stored on the TPM. Finally, the TPM can store hash values in Platform Configuration Registers, that may be used for attestation purposes.

TPM 2.0 implementations can come in various forms:

- Discrete TPMs are dedicated chips that implement TPM functionality in their own tamper resistant semiconductor package.
- Integrated TPMs are part of another chip.
- Firmware TPMs are software-only solutions that run in a CPU's trusted execution environment.
- Software TPMs are software emulators of TPMs that run with no more protection than a regular program gets within an operating system.
- Virtual TPMs are provided by a hypervisor and rely on the hypervisor to provide them with an isolated execution environment.

TPM technology has be criticized since it can be used to lock a device such that the owner of the device is prevented from installing certain software or from making certain changes to the existing software and device configuration. Ideally, the informed owner of a device should be able to take an informed decision whether she wants to trust the TPM embedded in a device. In reality, many owners will likely never ask this question and they may in fact may see benefits of trusting the vendor of a device (and if necessary be prepared to take legal action against the vendor if the local laws support that).



Trusted Execution Environment (TEE)

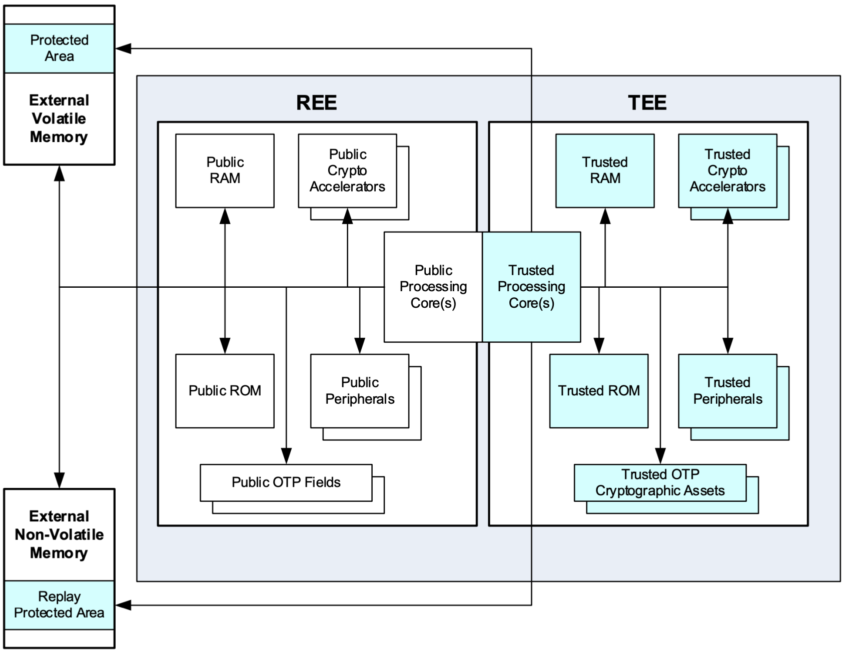
Definition (trusted and rich execution environment)

A *trusted execution environment* (TEE) is a secure area of a processor providing isolated execution, integrity of trusted applications, as well as confidentiality of trusted application resources. A *rich execution environment* (REE) is the non-secure area of a processor where an untrusted operating system executes.

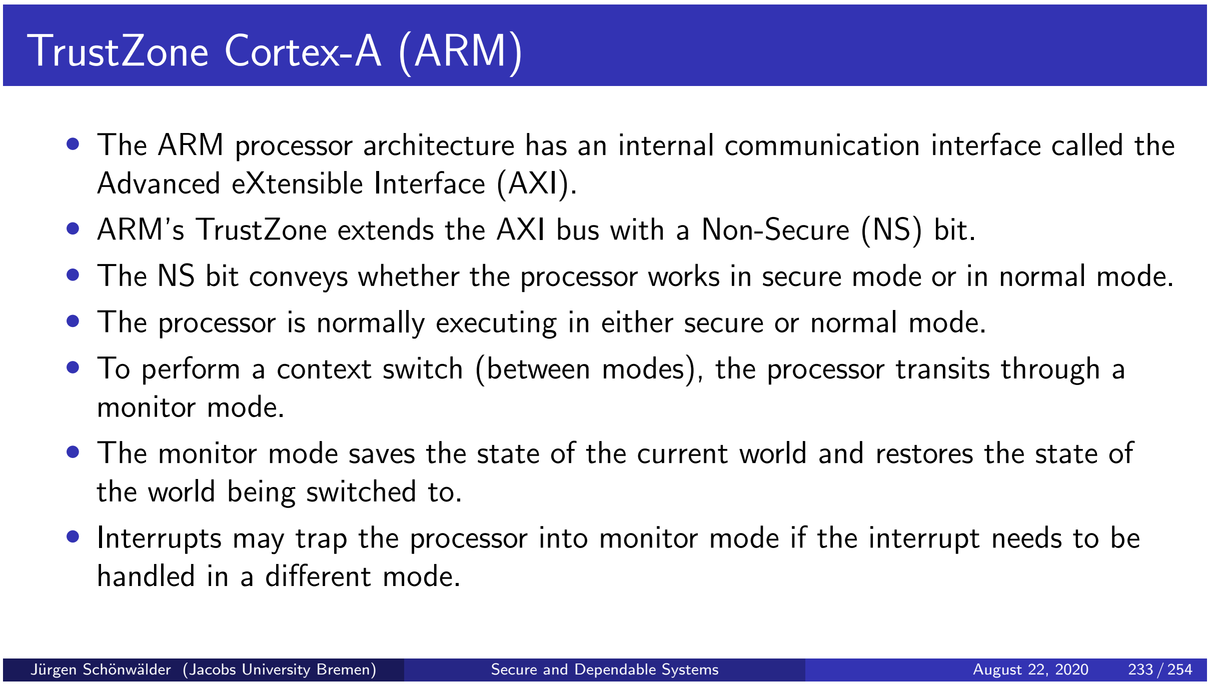
- REE resources are accessible from the TEE
- TEE resources are accessible from the REE only if explicitly allowed.
- The TEE specifications have been created by the GlobalPlatform (another industry consortium).

The TEE concept has been quite successful in the computing industry. There are several processor designs implementing TEEs and many mobile devices use TEE this technology to implement TPM-like functionality. On mobile phones, it is quite common these days that certain hardware components (e.g., a fingerprint reader) is only accessible to TEEs and the REE has to call the TEE to perform operation with the hardware. It is also common to implement secure boot technology using code running in the TEE in order to ensure that only authorized and untampered operating systems are loaded into the REE.

The architectural model provided in [21] is shown below:


Protected Area
External Volatile Memory
REE
TEE
Public RAM
Public Crypto Accelerators
Trusted RAM
Trusted Crypto Accelerators
Public Processing Core(s)
Trusted Processing Core(s)
Public ROM
Public Peripherals
Trusted ROM
Trusted Peripherals
Public OTP Fields
Trusted OTP Cryptographic Assets
External Non-Volatile Memory
Replay Protected Area

The software running inside a TEE is sometimes called trustlets and the TEE is occasionally called the “secure world” and the REE the “normal world”.



TrustZone Cortex-A (ARM)

- The ARM processor architecture has an internal communication interface called the Advanced eXtensible Interface (AXI).
- ARM's TrustZone extends the AXI bus with a Non-Secure (NS) bit.
- The NS bit conveys whether the processor works in secure mode or in normal mode.
- The processor is normally executing in either secure or normal mode.
- To perform a context switch (between modes), the processor transits through a monitor mode.
- The monitor mode saves the state of the current world and restores the state of the world being switched to.
- Interrupts may trap the processor into monitor mode if the interrupt needs to be handled in a different mode.

Jürgen Schönwälder (Jacobs University Bremen) Secure and Dependable Systems August 22, 2020 233 / 254

ARM has coined the term TrustZone but it resolves technically to two very different solution. The first solution, TrustZone for Cortex-A, is for relatively resource rich systems such as processors you find in your mobile phones. The second solution, TrustZone for Cortex-M, is for relatively resource limited systems such as processors that you find in embedded systems. There is often some confusion because people do not make the distinction between these two solutions explicit.

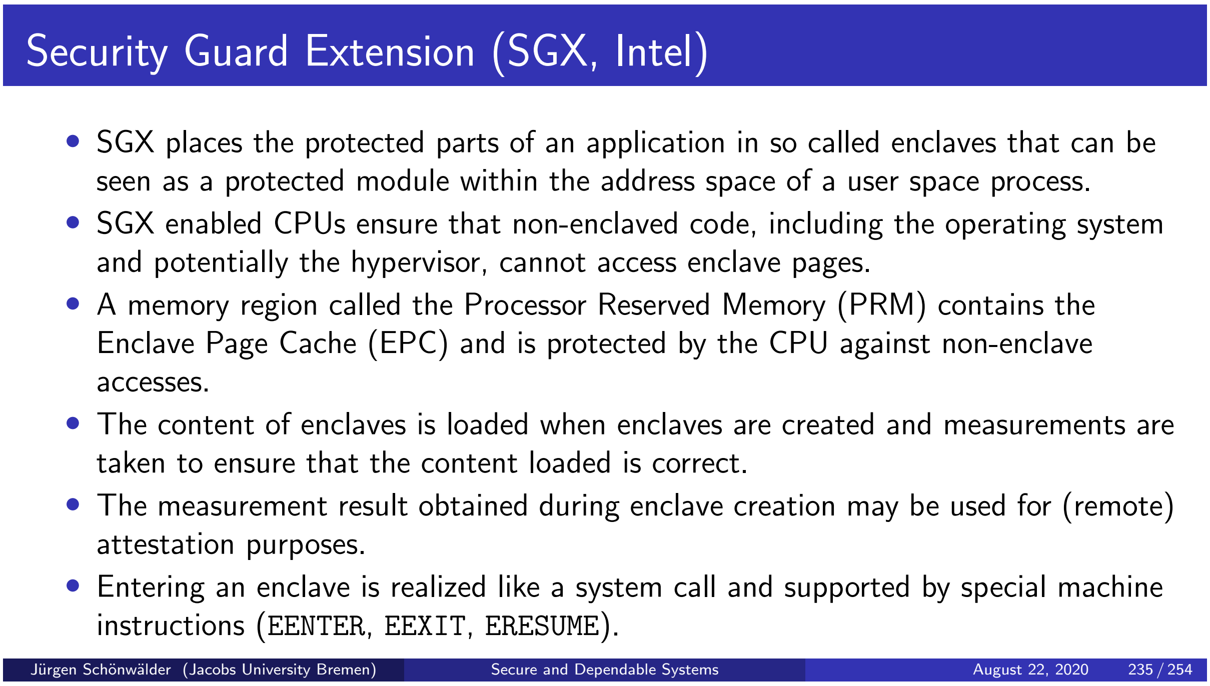
ARM's TrustZone architecture looks from a very high level like calls from user space programs into an operating system kernel implemented in hardware. The monitor mode is the entry point that carries out the mechanics of calling from normal mode into secure mode, ensuring proper isolation during the call.

TrustZone has been very successful in the mobile device market. Most of the operating systems executing on mobile devices do support TrustZone to implement TPM-like functionality and secure boot mechanisms. A detailed survey of TrustZone technology can be found in [38].

TrustZone Cortex-M (ARM)

- The Cortex-M design follows the Cortex-A design by having the processor execute in either secure or normal mode.
- Instructions read from secure memory will be executed in the secure mode of the processor and instructions read from non-secure memory will be executed in normal mode.
- Cortex-M replaces the monitor mode of the Cortex-A design with a faster mechanism to call secure code via multiple secure function entry points (supported by the machine instructions `SG`, `BXNS`, `BLXNS`).
- The Cortex-M design supports multiple separate call stacks and the memory space is separated into secure and non-secure sections.
- Interrupts can be configured to be handled in secure or non-secure mode.

TrustZone for Cortex-A has been introduced in 2004. The Cortex-M design is much newer and driven by the need to create trustworthy embedded systems. Cortex-M processors have no secure monitor mode and software. Instead, the transition between both worlds its handled by a set of mechanisms implemented into the core logic of the processor.



Security Guard Extension (SGX, Intel)

- SGX places the protected parts of an application in so called enclaves that can be seen as a protected module within the address space of a user space process.
- SGX enabled CPUs ensure that non-enclaved code, including the operating system and potentially the hypervisor, cannot access enclave pages.
- A memory region called the Processor Reserved Memory (PRM) contains the Enclave Page Cache (EPC) and is protected by the CPU against non-enclave accesses.
- The content of enclaves is loaded when enclaves are created and measurements are taken to ensure that the content loaded is correct.
- The measurement result obtained during enclave creation may be used for (remote) attestation purposes.
- Entering an enclave is realized like a system call and supported by special machine instructions (`EENTER`, `EEXIT`, `ERESUME`).

Intel SGX was introduced in 2015 with the sixth generation Intel Core processors. The design targets desktop and server platforms. It allows user-space processes to create private protected memory regions (the enclaves) that are isolated from other processes and also processes running at higher privilege levels (hypervisors or operating system kernels). Enclaves work almost transparently for existing hypervisors or memory management units.

If the capacity of the Enclave Page Cache (EPC) is exceeded, pages may be written to other memory regions after encrypting the content.

Creation and deletion of enclaves is performed by system software running at the highest privilege level while entering and leaving enclaves is done using the lowest privilege level.

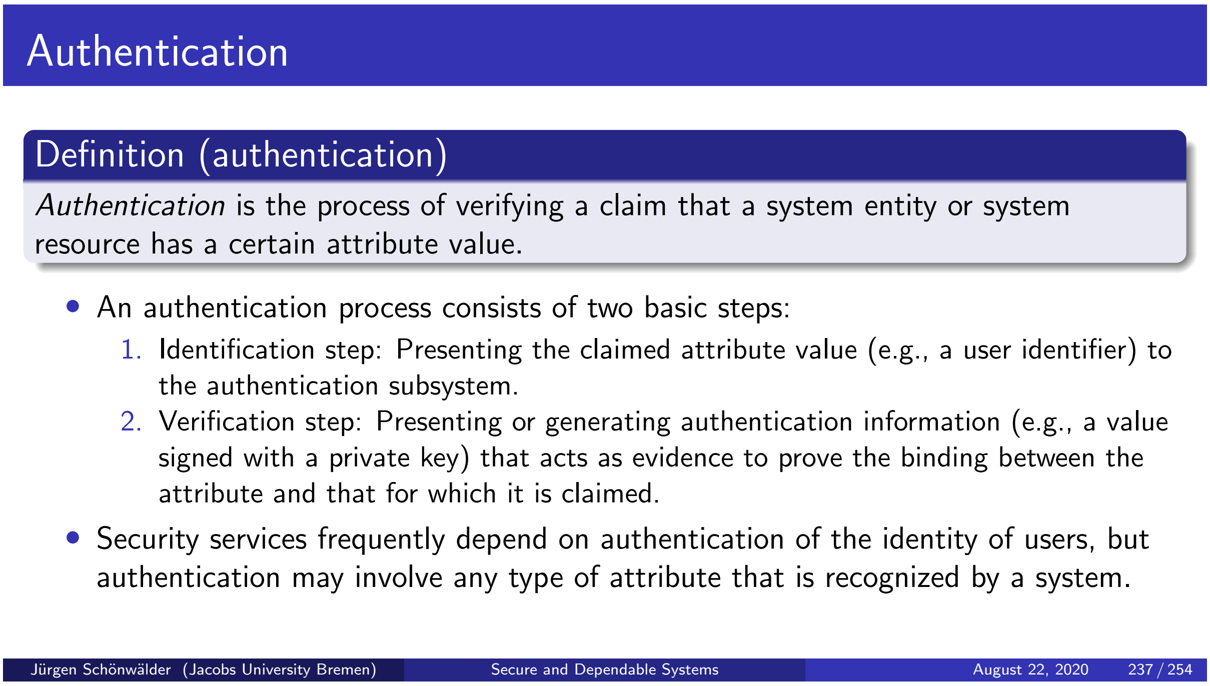
Authentication

29 Trusted Computing

30 Authentication

31 Authorization

32 Auditing



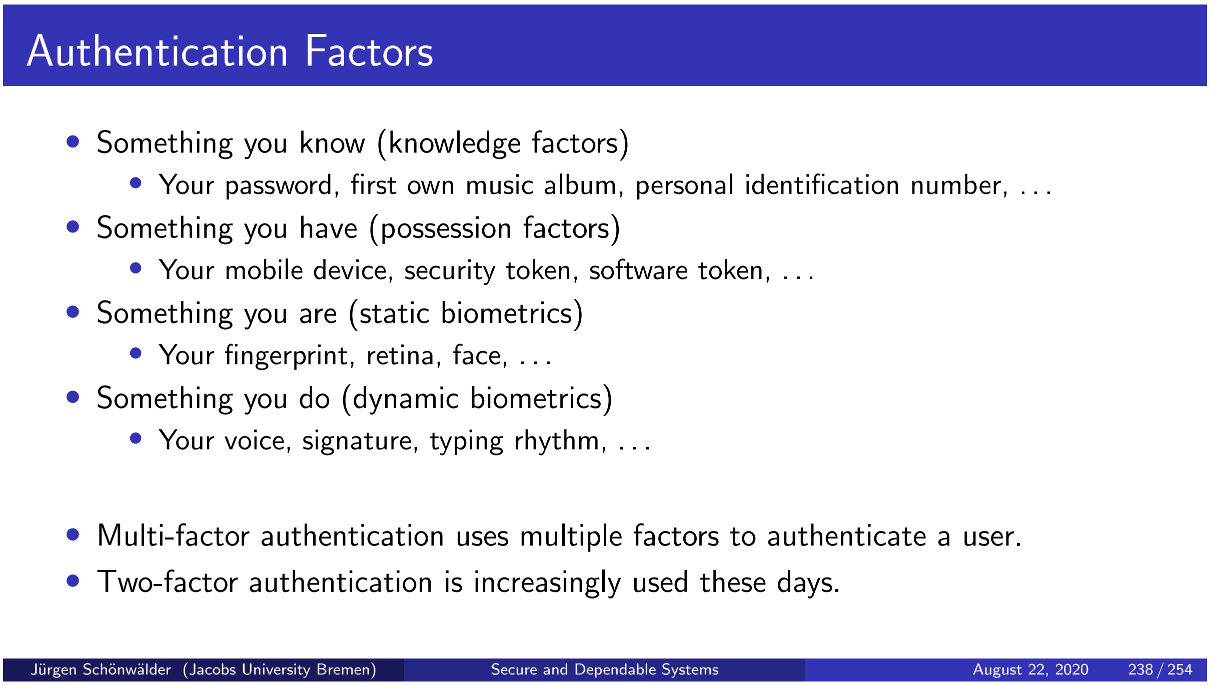
Authentication

Definition (authentication)

Authentication is the process of verifying a claim that a system entity or system resource has a certain attribute value.

- An authentication process consists of two basic steps:
  1. Identification step: Presenting the claimed attribute value (e.g., a user identifier) to the authentication subsystem.
  2. Verification step: Presenting or generating authentication information (e.g., a value signed with a private key) that acts as evidence to prove the binding between the attribute and that for which it is claimed.
- Security services frequently depend on authentication of the identity of users, but authentication may involve any type of attribute that is recognized by a system.

This definition is taken from RFC 4949 [46].



Authentication Factors

- Something you know (knowledge factors)
  - Your password, first own music album, personal identification number, . . .
- Something you have (possession factors)
  - Your mobile device, security token, software token, . . .
- Something you are (static biometrics)
  - Your fingerprint, retina, face, . . .
- Something you do (dynamic biometrics)
  - Your voice, signature, typing rhythm, . . .

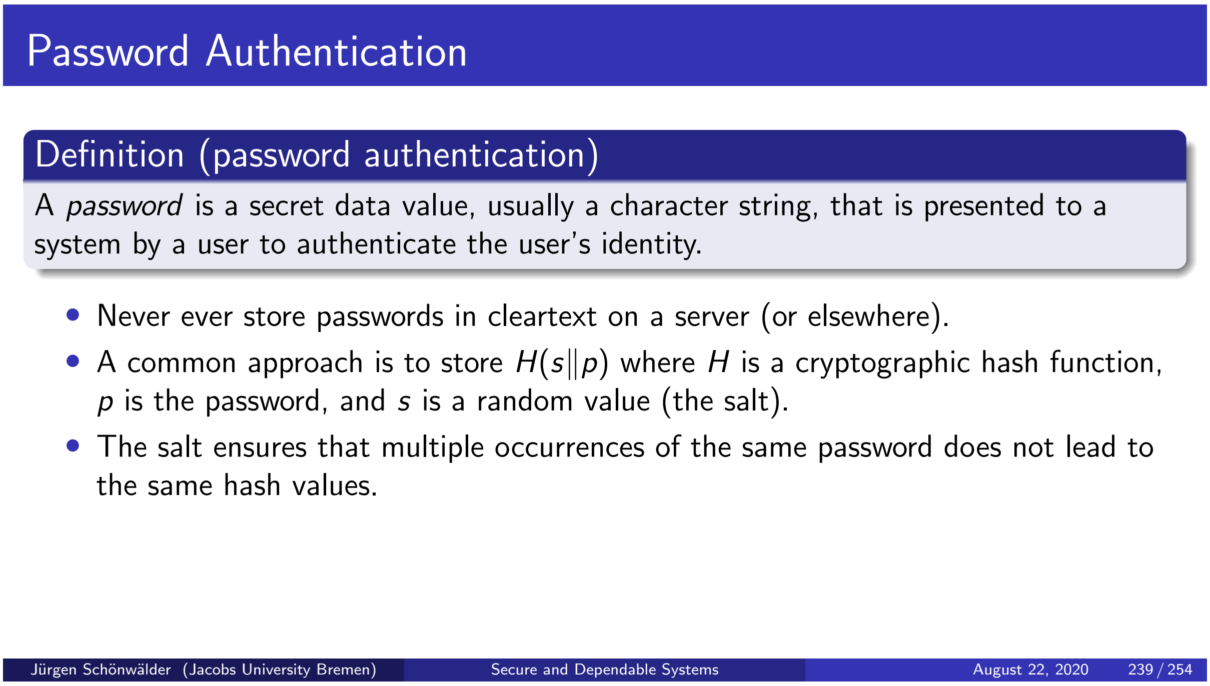
- Multi-factor authentication uses multiple factors to authenticate a user.
- Two-factor authentication is increasingly used these days.

Humans are not very good at remembering good security tokens. The use of passwords has cause many security problems. It is fairly easy to write programs that can try out large lists of popular passwords or search through a significant portion of the password space. Forcing users to create “stronger” passwords often leads to undesirable side effects, such as users writing down passwords on stickers. Ideally, passwords are used rarely to unlock other cryptographic keys that subsequently are used for authentication purposes.

Inherent factors were often shown in early science fiction movies but meanwhile mobile devices use bio-metrics to identify the owner of a device. An inherent downside of bio-metrics is that they can’t be changed. If someone “steals” your fingerprint and then produces a fake finger that tricks a fingerprint reader into believing that there is a real finger, then you have a serious problem since you can simply change your fingerprint.

The downside of possession factors is simply that they may get lost or stolen. If you loose the key to your house, you may have to replace all locks in order to prevent someone to enter your house with the lost key.

The bottom line is that all techniques have their specific downsides. By combining multiple authentication techniques, it is possible to significantly raise the strengths of the authentication system and to reduce the impact of the downsides of the authentication techniques used.



Password Authentication

Definition (password authentication)

A *password* is a secret data value, usually a character string, that is presented to a system by a user to authenticate the user's identity.

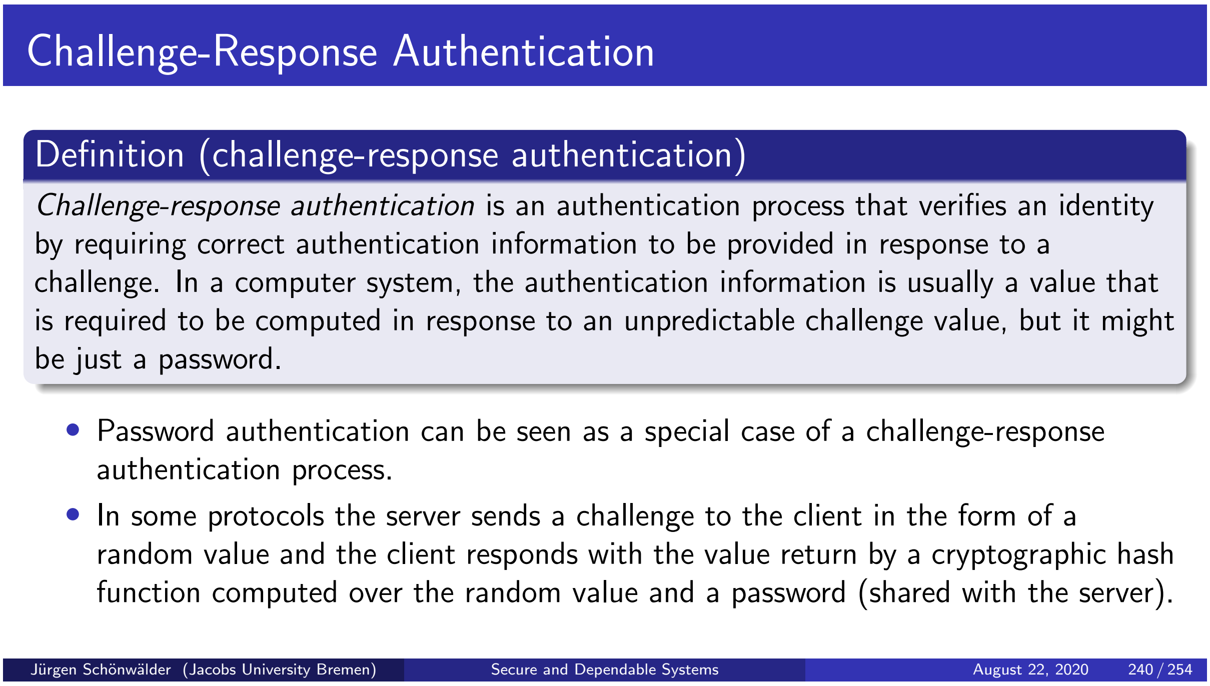
- Never ever store passwords in cleartext on a server (or elsewhere).
- A common approach is to store $H(s||p)$ where H is a cryptographic hash function, p is the password, and s is a random value (the salt).
- The salt ensures that multiple occurrences of the same password does not lead to the same hash values.

Passwords have been a big source of security problems but they continue to be popular. Typical attacks on passwords:

- offline dictionary attacks
- online dictionary attack
- attacks with large collections of popular passwords
- user mistakes (writing passwords down)
- tricking users to share passwords (phishing)
- exploiting multiple password use (users tend to reuse passwords)
- eavesdropping (key logger, cameras, insecure network protocols)

The advice to add special symbols to passwords often does not lead to the desired effects and they may force humans to write passwords down (without proper protection of the paper).

xkcd: Password Strength



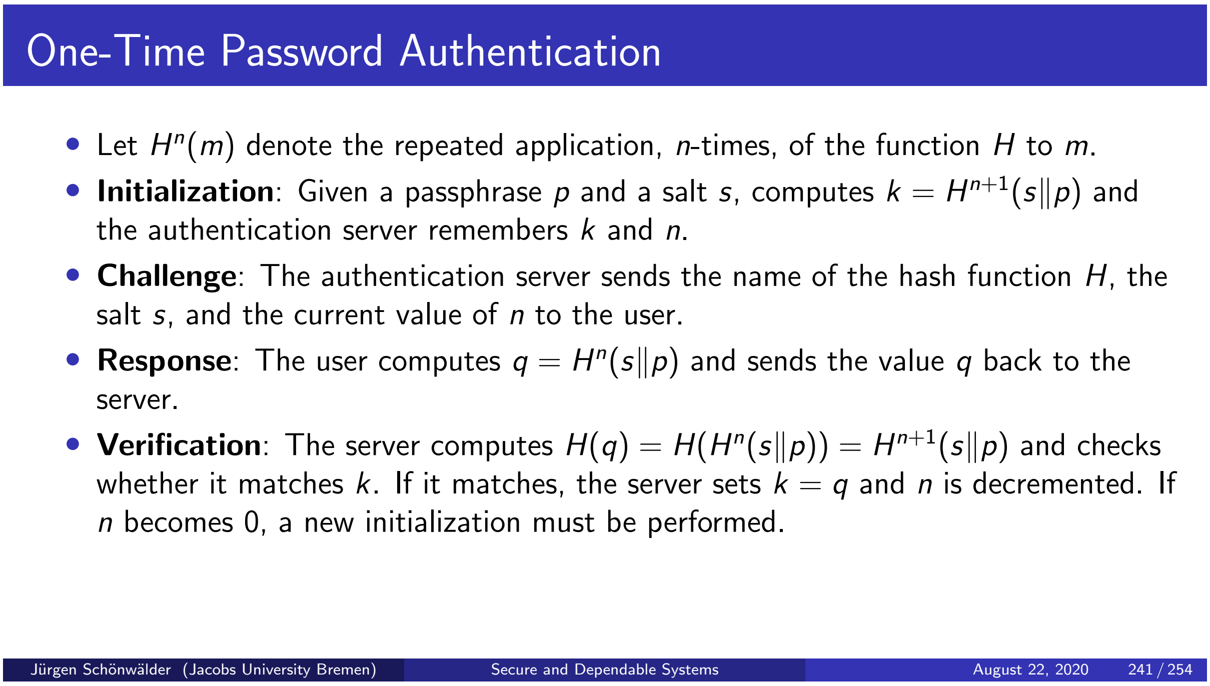
Challenge-Response Authentication

Definition (challenge-response authentication)

Challenge-response authentication is an authentication process that verifies an identity by requiring correct authentication information to be provided in response to a challenge. In a computer system, the authentication information is usually a value that is required to be computed in response to an unpredictable challenge value, but it might be just a password.

- Password authentication can be seen as a special case of a challenge-response authentication process.
- In some protocols the server sends a challenge to the client in the form of a random value and the client responds with the value return by a cryptographic hash function computed over the random value and a password (shared with the server).

The main motivation for challenge-response authentication mechanism is that they can avoid sending a cleartext password. A downside is that the server may need access to the cleartext password to verify the response returned by the authenticating client. Challenge-response protocols where popular in early network dialin protocols.

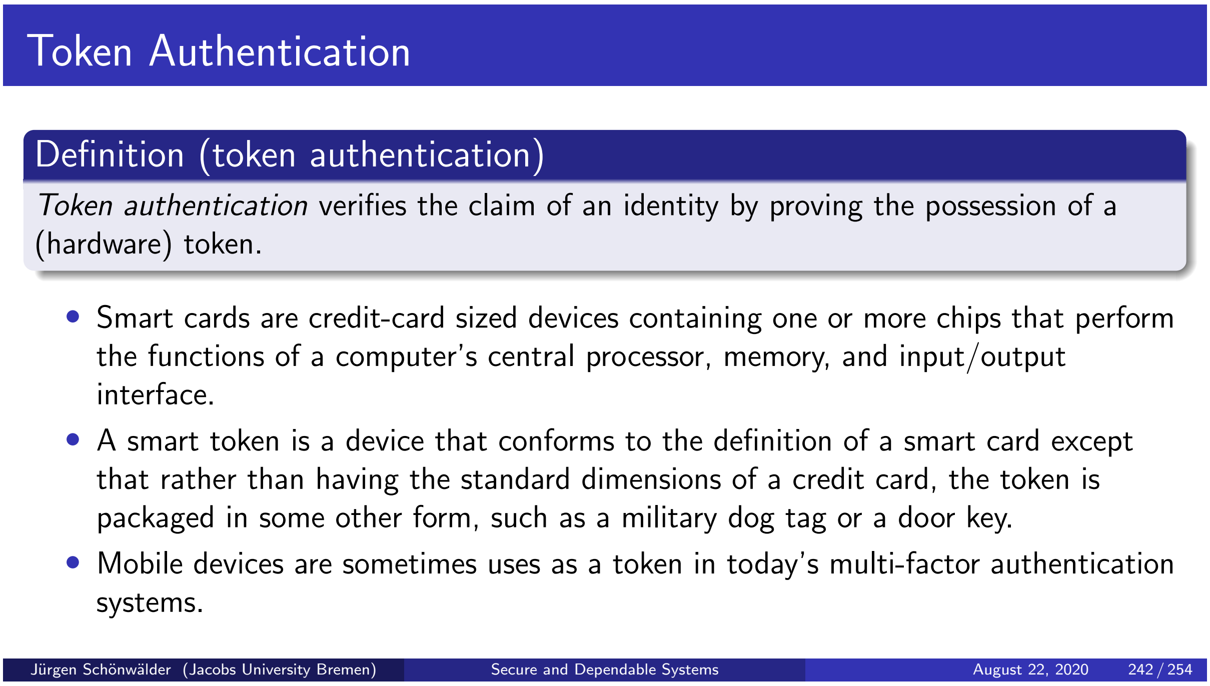


One-Time Password Authentication

- Let $H^n(m)$ denote the repeated application, n-times, of the function H to m.
- **Initialization**: Given a passphrase p and a salt s, computes $k = H^{n+1}(s\|p)$ and the authentication server remembers k and n.
- **Challenge**: The authentication server sends the name of the hash function H, the salt s, and the current value of n to the user.
- **Response**: The user computes $q = H^n(s\|p)$ and sends the value q back to the server.
- **Verification**: The server computes $H(q) = H(H^n(s\|p)) = H^{n+1}(s\|p)$ and checks whether it matches k. If it matches, the server sets $k = q$ and n is decremented. If n becomes 0, a new initialization must be performed.

One-time passwords are described in RFC 2289 [**?**]. Given a cryptographic hash function H, it is easy to compute H^n from H^{n-1}, but it is difficult to compute H^{n-1} from H^n. Note that the server does not store the password nor is the response to the challenge valid more than once. This makes the system robust against replay attacks.

According to RFC 4949 [46], a *one-time password* is a simple authentication technique in which each password is used only once as authentication information that verifies an identity. This technique counters the threat of a replay attack that uses passwords captured by wiretapping.



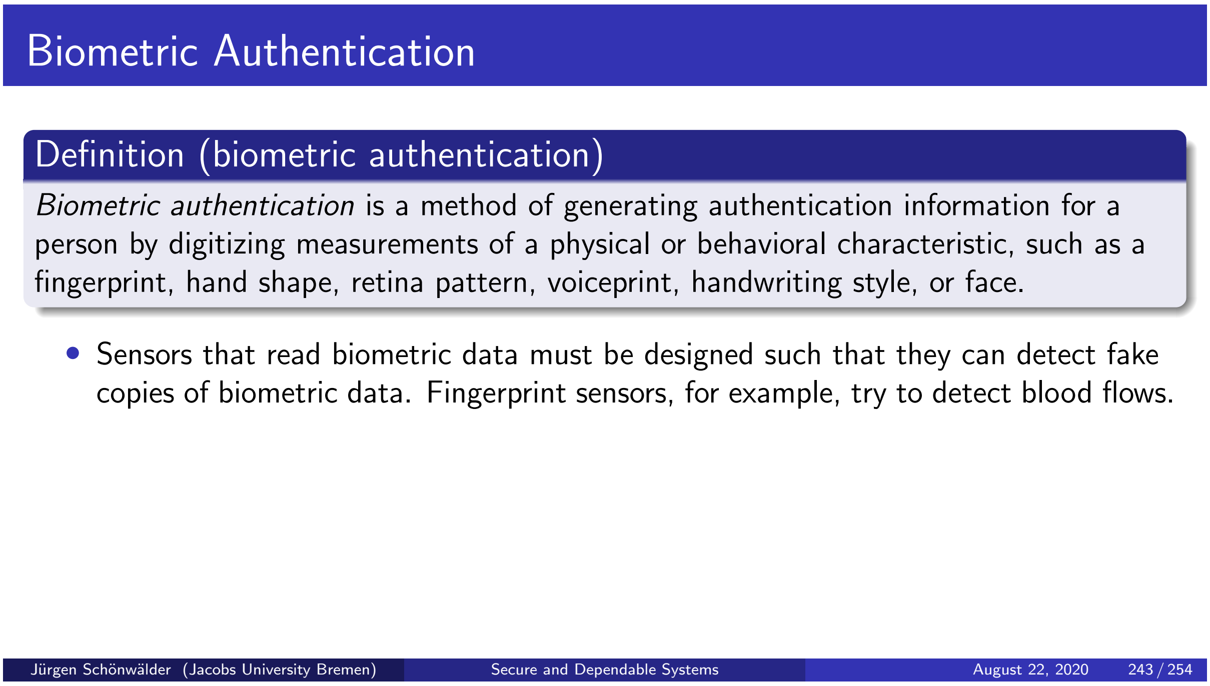
Token Authentication

Definition (token authentication)

Token authentication verifies the claim of an identity by proving the possession of a (hardware) token.

- Smart cards are credit-card sized devices containing one or more chips that perform the functions of a computer's central processor, memory, and input/output interface.
- A smart token is a device that conforms to the definition of a smart card except that rather than having the standard dimensions of a credit card, the token is packaged in some other form, such as a military dog tag or a door key.
- Mobile devices are sometimes uses as a token in today's multi-factor authentication systems.

An advantage of digital tokens is that they resemble traditional keys, something most people are used to. A downside of digital tokens is that they resemble traditional keys, they may be passed on to others, they may get lost or stolen, and they may get cloned.



Biometric Authentication

Definition (biometric authentication)

Biometric authentication is a method of generating authentication information for a person by digitizing measurements of a physical or behavioral characteristic, such as a fingerprint, hand shape, retina pattern, voiceprint, handwriting style, or face.

- Sensors that read biometric data must be designed such that they can detect fake copies of biometric data. Fingerprint sensors, for example, try to detect blood flows.

Biometric data has privacy concerns and, depending on your jurisdiction, may be subject to strong data protection laws.

A fundamental problem of biometric authentication mechanisms is that the measured biological characteristic cannot be changed.

Authorization

29 Trusted Computing

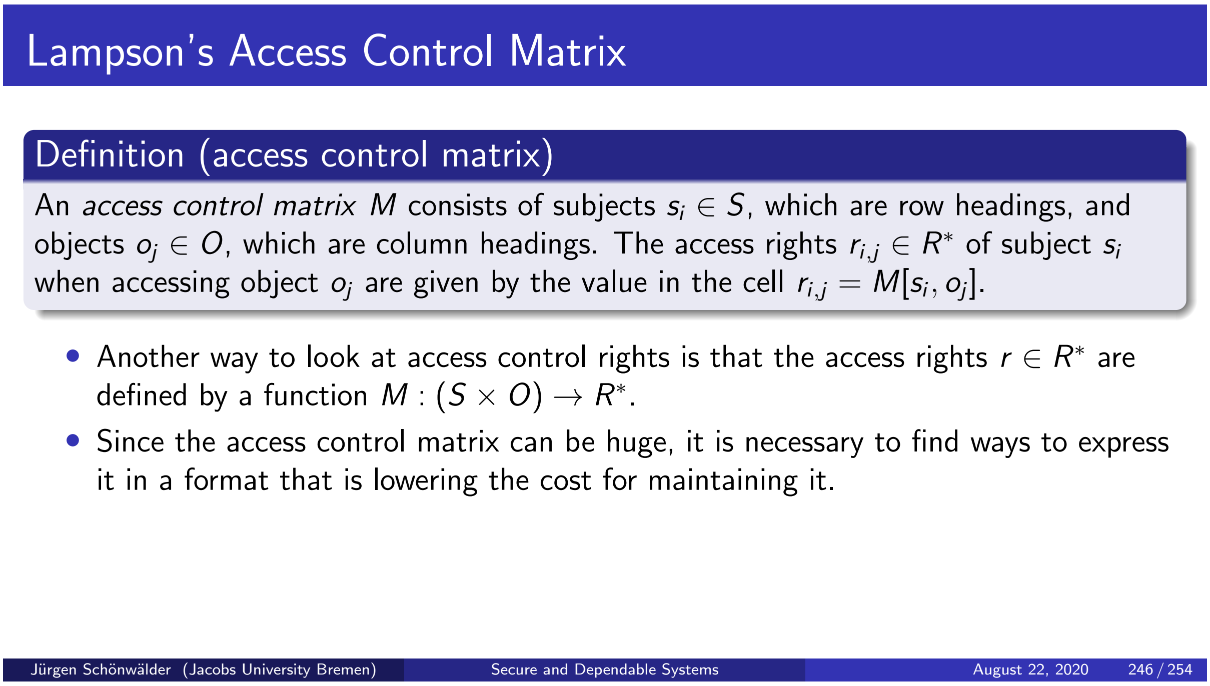
30 Authentication

31 Authorization

32 Auditing

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

Definition (access control matrix)

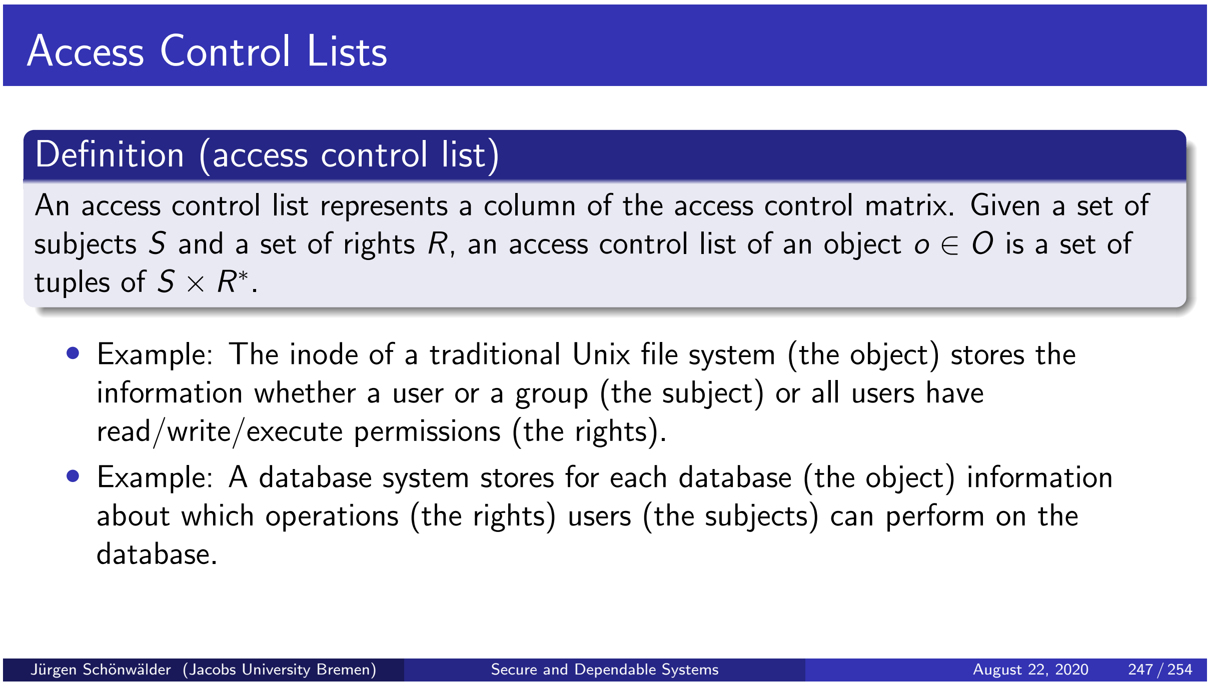
An *access control matrix* M consists of subjects $s_i \in S$, which are row headings, and objects $o_j \in O$, which are column headings. The access rights $r_{i,j} \in R^*$ of subject s_i when accessing object o_j are given by the value in the cell $r_{i,j} = M[s_i, o_j]$.

- Another way to look at access control rights is that the access rights $r \in R^*$ are defined by a function $M : (S \times O) \rightarrow R^*$.
- 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 but 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.

Here is an example access control matrix:

	moodle	wifi	printer
Alice		access	print
Bob	student	access	
Carol	instructor	access	print, scan
Charlie		access	scan
Dave	student	access	



Access Control Lists

Definition (access control list)

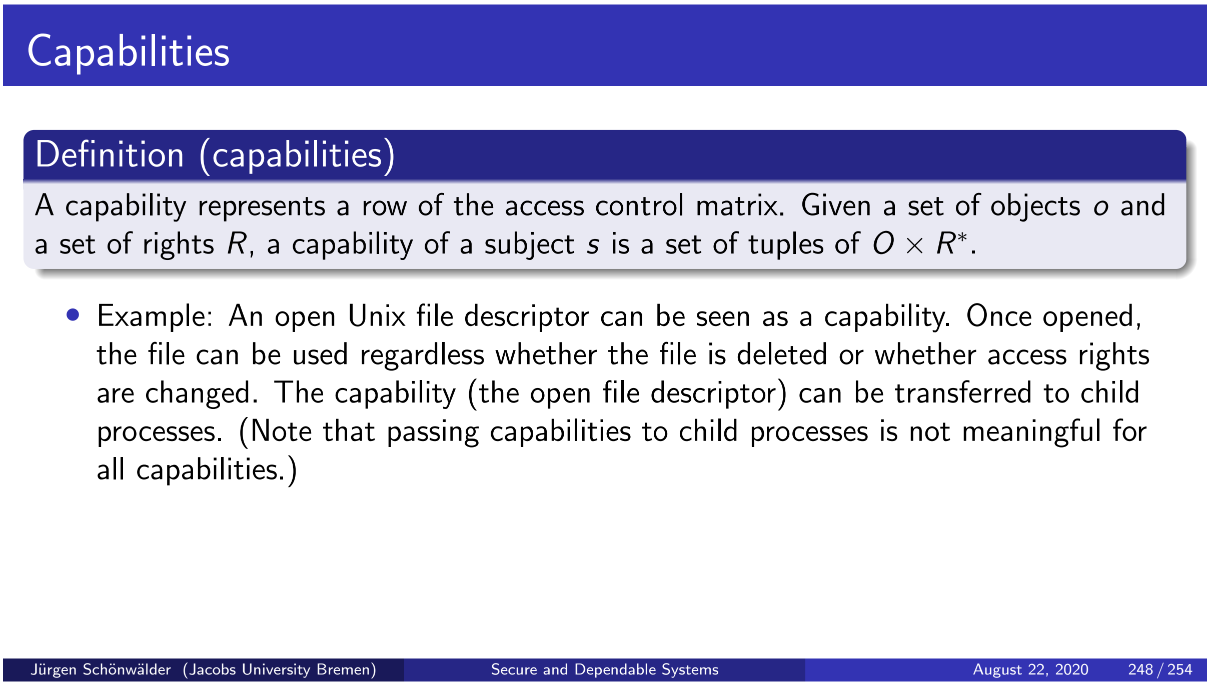
An access control list represents a column of the access control matrix. Given a set of subjects S and a set of rights R, an access control list of an object $o \in O$ is a set of tuples of $S \times R^*$.

- Example: The inode of a traditional Unix file system (the object) stores the information whether a user or a group (the subject) or all users have read/write/execute permissions (the rights).
- Example: A database system stores for each database (the object) information about which operations (the rights) users (the subjects) can perform on the database.

Typical 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?

ACLs can become very complicated and difficult to manage. A good example are network packet filters where the ACL consists of long chains of rules that over time become very difficult to maintain.



Capabilities

Definition (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 transferred to child processes. (Note that passing capabilities to child processes is not meaningful for all capabilities.)

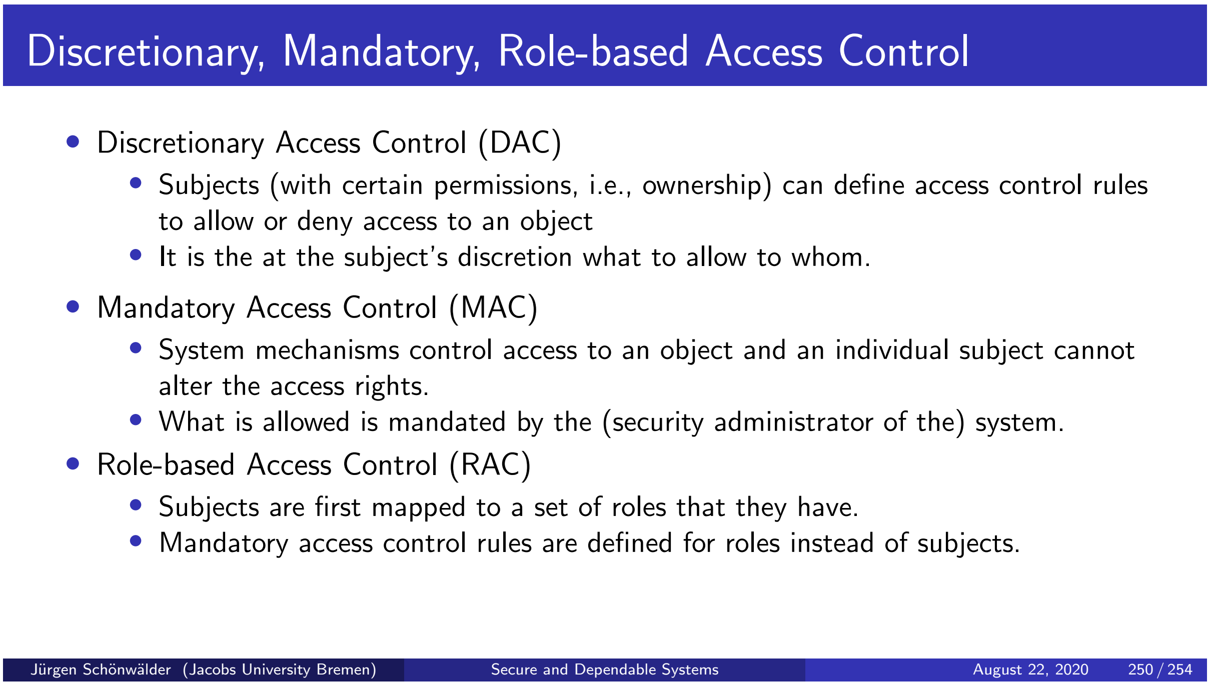
Capabilities are like tickets that allow a subject to do certain things. It is essential that subjects cannot alter their capabilities in an uncontrolled way. Operating systems therefore typically maintain capabilities in kernel space. The file descriptor, for example, is maintained in the kernel and it cannot be changed to refer to a different file without the involvement of the kernel.

Typical design issues for capabilities:

- 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.



Discretionary, Mandatory, Role-based Access Control

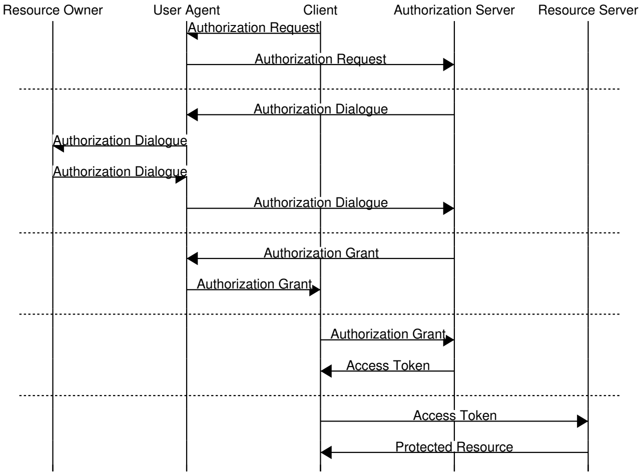
- Discretionary Access Control (DAC)
  - Subjects (with certain permissions, i.e., ownership) can define access control rules to allow or deny access to an object
  - It is the at the subject's discretion what to allow to whom.
- Mandatory Access Control (MAC)
  - System mechanisms control access to an object and an individual subject cannot alter the access rights.
  - What is allowed is mandated by the (security administrator of the) system.
- Role-based Access Control (RAC)
  - Subjects are first mapped to a set of roles that they have.
  - Mandatory access control rules are defined for roles instead of subjects.

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 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.

API Authorization (OAuth 2.0)



OAuth 2.0 [1] is a protocol used by a client to obtain authorization to access certain remote resources (typically APIs accessible via HTTP) on the Internet. The protocol involves a Resource Owner (for example a user owning an resource), a Client (for example an application on a backend server) interested to access a certain resource, an Authorization Server authorizing access to a resource (by generating an Access Token), and the Resource Server (a server providing access to the resource).

The slide shows a typical workflow that involves a User Agent (typically a web browser). The Client starts the process by sending an authorization request via the users' browser to the Authorization Server. The request includes information about the scope of the access. The Authorization Server then interacts with the Resource Owner (the user of the web browser in this scenario) to obtain the permissions to grant the access. Once permissions have been obtained, the Authorization Server generates an Authorization Grant that is returned via the User Agent to the Client. The Client uses the Access Grant to obtain an Authorization Token from the Authorization Server. This communication is between two backend systems and does not involve the User Agent. Once the Client receives an Authorization Token, it can access the resource on the Resource Server by passing along the Authorization Token. The Authorization Token has a limited lifetime and the Client may have to obtain a new Authorization Token if the current token has expired.

YouTube: OAuth 2.0 and OpenID Connect (in plain English)

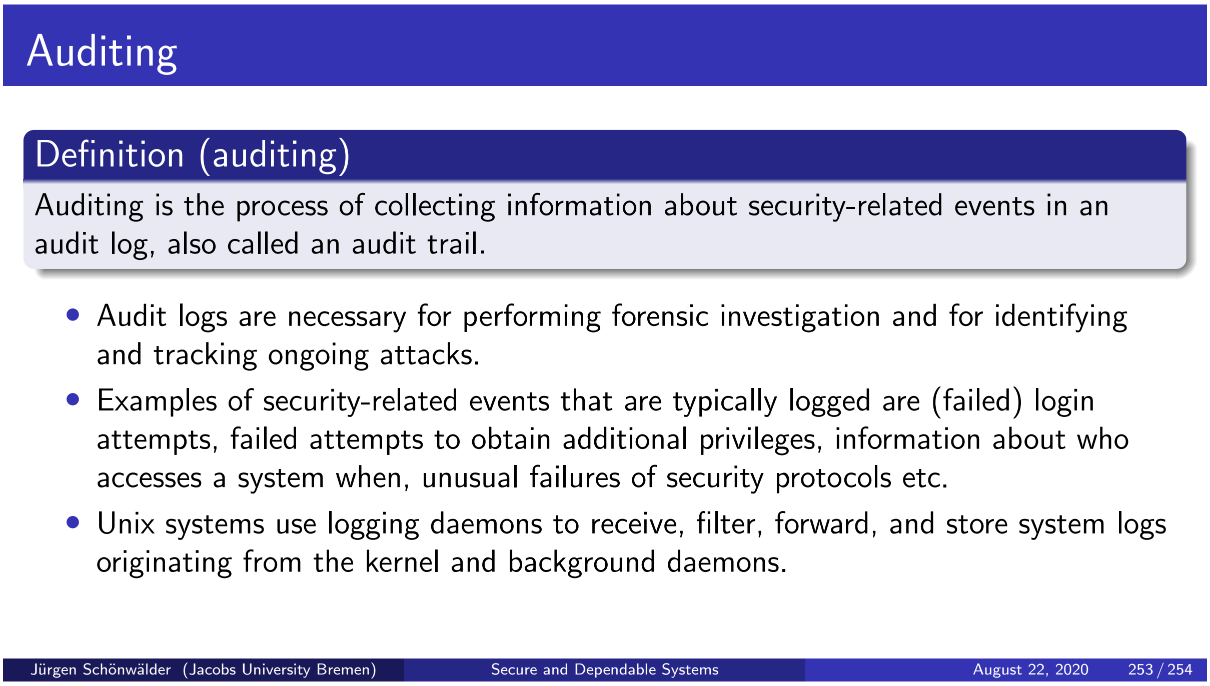
Auditing

29 Trusted Computing

30 Authentication

31 Authorization

32 Auditing



Auditing

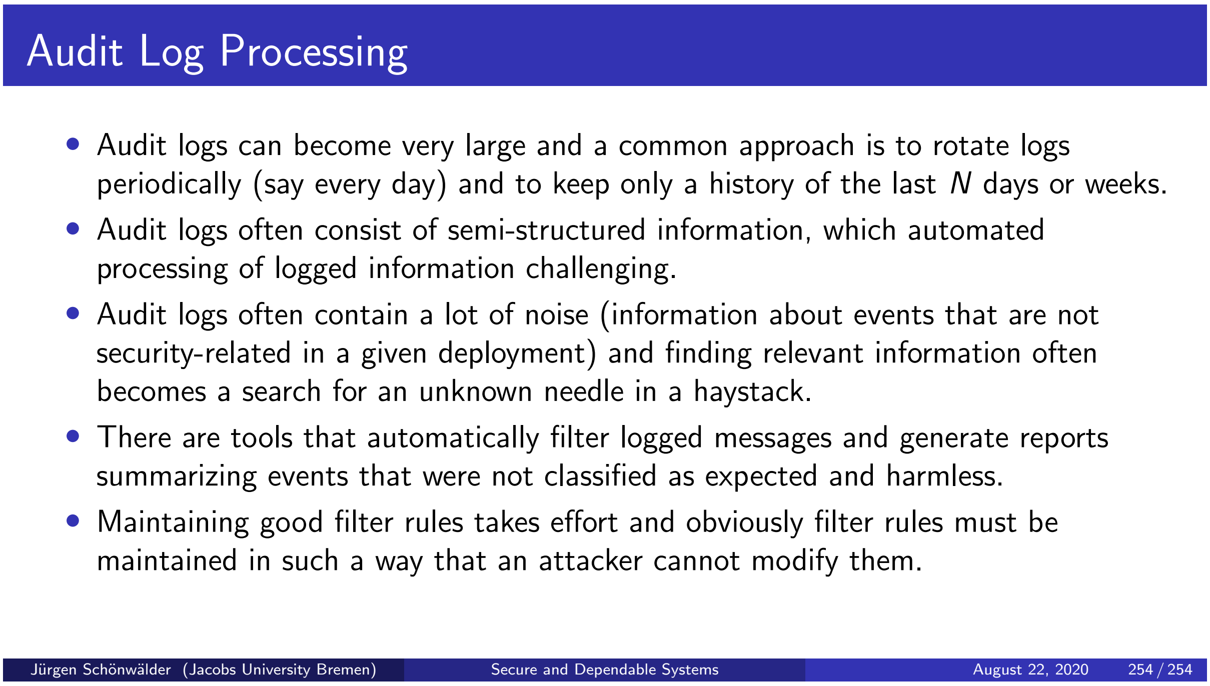
Definition (auditing)

Auditing is the process of collecting information about security-related events in an audit log, also called an audit trail.

- Audit logs are necessary for performing forensic investigation and for identifying and tracking ongoing attacks.
- Examples of security-related events that are typically logged are (failed) login attempts, failed attempts to obtain additional privileges, information about who accesses a system when, unusual failures of security protocols etc.
- Unix systems use logging daemons to receive, filter, forward, and store system logs originating from the kernel and background daemons.

It is essential to collect information about any security-related events that have been detected by an operating system or application software. For example, repeated failed login attempts may indicate that a password guessing attack is going on. Logs of security-related events can also provide valuable information after a system has been attacked in order to understand how an attack was performed (forensics).

Since security event logs pose a risk for an attacker, an attacker may be interested to modify event logs as part of the attack. Hence it is necessary to protect the integrity of event logs. Furthermore, it may be necessary to be able to prove in front of a court that a security event log was obtained from a specific system, i.e., that the security log is authentic. As a consequence, it is desirable that security event logs are (i) not stored on the devices themself, (ii) integrity protected, and (iii) properly signed by a key that is bound to the device.



Audit Log Processing

- Audit logs can become very large and a common approach is to rotate logs periodically (say every day) and to keep only a history of the last N days or weeks.
- Audit logs often consist of semi-structured information, which automated processing of logged information challenging.
- Audit logs often contain a lot of noise (information about events that are not security-related in a given deployment) and finding relevant information often becomes a search for an unknown needle in a haystack.
- There are tools that automatically filter logged messages and generate reports summarizing events that were not classified as expected and harmless.
- Maintaining good filter rules takes effort and obviously filter rules must be maintained in such a way that an attacker cannot modify them.

A simple but effective filtering tool is `logcheck`. It uses a (often pretty large) collection of regular expressions to filter log messages.

Much more advanced tools exist such as the Elastic Stack, still commonly known has the ELK (Elastic Search, Logstash, Kibana) stack.

- Logstash is a tool that parses semi-structured log information and generates structured reports often enriched with additional information.
- The data generated by logstash can be fed into a scalable search engine such as Elastic Search, which stores the data and is able to execute search queries efficiently on the stored data.
- Kibana is a dashboard frontend for Elastic Search that can generate several different representations and visualizations of data stored by Elastic Search.

References

[1] The OAuth 2.0 Authorization Framework. RFC 6749, Microsoft, October 2012.

[2] S. Turner A. Langley, M. Hamburg. Elliptic Curves for Security. RFC 7748, Google, Rambus Cryptography Research, sn3rd, January 2016.

[3] M. Abadi and D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1), January 1999.

[4] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Proc. 2nd International Workshop on Information Hiding*, pages 73–82. Springer, 1998.

[5] 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.

[6] 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.

[7] Austen Barker, Staunton Sample, Yash Gupta, Anastasia McTaggart, Ethan L. Miller, and Darrell D. E. Long. Artifice: A deniable steganographic file system. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*, Santa Clara, CA, August 2019. USENIX Association.

[8] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. Automatic Certificate Management Environment (ACME). RFC 8555, Cisco, EFF, Let's Encrypt, University of Michigan, March 2019.

[9] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008.

[10] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, page 77–84, New York, NY, USA, 2004. Association for Computing Machinery.

[11] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Operating Systems Review*, 23(5):1–13, 1989.

[12] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880, PGP Corporation, IKS GmbH, November 2007.

[13] D.L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, NIST, Microsoft, Trinity College Dublin, Entrust, Vigil Security, May 2008.

[15] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.

[16] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Independent, RTFM, August 2008.

[17] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 6:644–654, November 1976.

[18] 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.

[19] 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.

[20] D. Eastlake and T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, Huawei, AT&T Labs, May 2011.

[21] Global Platform. TEE System Architecture Version 1.2. GlobalPlatform Technology GPD_SPE_009, Global Platform, November 2018.

[22] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[23] R. Housley. Cryptographic Message Syntax. RFC 5652, Vigil Security, September 2009.

[24] M. Jones, J. Bradley, and N. Sakimura. JSON Web Signature (JWS). RFC 7515, Microsoft, Ping Identity, NRI, May 2015.

[25] B. Kaduk and M. Short. Deprecate Triple-DES (3DES) and RC4 in Kerberos. RFC 8429, Akamai, Microsoft Corporation, October 2018.

[26] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 1995.

[27] A. Kehne, J. Schönwälder, and H. Langendörfer. A Nonce-Based Protocol for Multiple Authentications. *ACM Operating System Review*, 26(4):84–89, October 1992.

[28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[29] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 310–331, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[30] B.W. Lampson. Computer Security in the Real World. *IEEE Computer*, 37(6):37–46, June 2004.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[32] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[33] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for linux. In *Proc. 3rd International Workshop on Information Hiding*, pages 463–477. Springer, 1999.

[34] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, Cisco Systems, January 2008.

[35] N. Modadugu and E. Rescorla. The Design and Implementation of Datagram TLS. In *Proc. Network and Distributed System Security Symposium*, San Diego, February 2004.

[36] 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.

[37] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120, USC-ISI, MIT, July 2005.

[38] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), January 2019.

[39] N. Provos and P. Honeyman. Hide and Seek: An Introduction to Steganography. *IEEE Security and Privacy*, 1(3), June 2003.

[40] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Mozilla, August 2018.

[41] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 6347, RTFM, Google, January 2012.

[42] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key-cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[43] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125, Cisco, PayPal, March 2011.

[44] J. Schaad. CBOR Object Signing and Encryption (COSE). RFC 8152, August Cellars, August 2017.

[45] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457, Porticor, Technische Universitaet Muenchen, &yet, February 2015.

[46] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, August 2007.

[47] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Comput. Surv.*, 49(3), October 2016.

[48] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[49] S. Wendzel W. Mazurczyk. Information Hiding: Challenges for Forensic Experts. *Communications of the ACM*, 61(1):86–94, January 2018.

[50] 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.

[51] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, SSH Communications Security Corp, Cisco Systems, January 2006.

[52] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254, SSH Communications Security Corp, Cisco Systems, January 2006.

[53] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, SSH Communications Security Corp, Cisco Systems, January 2006.

[54] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, SSH Communications Security Corp, Cisco Systems, January 2006.