Secure and Dependable Systems

Jürgen Schönwälder

Jacobs University Bremen

June 11, 2019

https://cnds.jacobs-university.de/courses/sads-2019

Part: Administrivia

1. Course Objectives and Grading

2. Rules of the Game

3. Resources

Course Objectives and Grading

1. Course Objectives and Grading
2. Rules of the Game
3. Resources

Topics and learning goals

- This course introduces (formal) methods for analyzing and assuring safety and security of software systems.
- Definition of concepts such as dependability, reliability, safety, and security of software systems
- Introduction of paradigms of safety/security analysis such as
  - formal testing (code coverage),
  - static program analysis (control/data flow analysis and abstract interpretation),
  - model checking (computational tree logic),
  - and program verification (Hoare calculus).
- Introduction into cryptography and its application for building secure systems.

Grading Scheme

- Assignments (30%)
  - Learning by solving assignments
  - Test whether you can apply concepts learned
- Mid-term examination (30%)
  - Covers the first half of the course
  - Closed book (and closed computers)
- Final examination (40%)
  - Covers the whole course
  - Closed book (and closed computers)

Teaching and learning strategy

- Homework assignments: Reinforce and apply what is taught in class
- Assignments will be small individual assignments (but may take time to solve)
- Consider forming study groups. It helps to discuss questions and course material in study groups or to explore different directions to solve an assignment. However, solutions must be individual submissions. (Discuss the general problem in a study group, workout the details of the solution yourself.)
- You can audit the course. To earn an audit, you have to pass an oral interview about key concepts introduced in the course at the end of the semester.

Organizational aspects and tutorials

- All homework assignments will be linked to the course web page.
- Solutions for assignments will be submitted using the Moodle system. `https://moodle.jacobs-university.de/`
- Feedback and grades will be accessible via the Moodle system as well.
- Teaching assistant will be available to discuss course topics and or questions related to homeworks or to get help during exam preparations.

Rules of the Game

1 Course Objectives and Grading

2 Rules of the Game

3 Resources

Code of Academic Integrity

- Jacobs University has a “Code of Academic Integrity”
  - this is a document approved by the Jacobs community
  - you have signed it during enrollment
  - it is a law of the university, we take it seriously
- It mandates good behaviours from faculty and students and it penalizes bad ones:
  - honest academic behavior (e.g., no cheating)
  - respect and protect intellectual property of others (e.g., no plagiarism)
  - treat all Jacobs University members equally (e.g., no favoritism)
- It protects you and it builds an atmosphere of mutual respect
  - we treat each other as reasonable persons
  - the other’s requests and needs are reasonable until proven otherwise
  - if others violate our trust, we are deeply disappointed (may be leading to severe and uncompromising consequences)

Academic Integrity Committee (AIC)

- The Academic Integrity Committee is a joint committee by students and faculty.
- Mandate: to hear and decide on any major or contested allegations, in particular,
    - the AIC decides based on evidence in a timely manner,
    - the AIC makes recommendations that are executed by academic affairs,
    - the AIC tries to keep allegations against faculty anonymous for the student.
- Every member of Jacobs University (faculty, student, ...) can appeal any academic integrity allegations to the AIC.

Cheating

- There is no need to cheat, cheating prevents you from learning.
- Useful collaboration versus cheating:
  - You will be required to hand in your own original code/text/math for all assignments
  - You may discuss your homework assignments with others, but if doing so impairs your ability to write truly original code/text/math, you will be cheating
  - Copying from peers, books or the Internet is plagiarism unless properly attributed
- What happens if we catch you cheating?
  - We will confront you with the allegation (you can explain yourself)
  - If you admit or are silent, we impose a grade sanction and we notify the student records office
  - Repeated infractions to go the AIC for deliberation
- Note: Both active (copying from others) and passive cheating (allowing others to copy) are penalized equally

Deadlines

- Deadlines will be strict (don't bother to ask for extensions)
- Make sure you submit the right document. We grade what was submitted, not what could have been submitted.
- Submit early - avoid last minute changes or software/hardware problems.
- Official excuses by the student records office will extend the deadlines - but not more than the time covered by the excuse.
- A word on medical excuses: Use them when you are ill. Do not use them as a tool to gain more time.
- You want to be taken serious if you are seriously ill. Misuse of excuses can lead to a situation where you are not taken very serious when you deserve to be taken serious.

Culture of Questions, Answers, and Explanations

- Answers to questions require an explanation even if this is not stated explicitly
    - A question like 'Does this algorithm always terminate?' can in principle be answered with 'yes' or 'no'.
    - We expect, however, that an explanation is given why the answer is 'yes' or 'no', even if this is not explicitly stated.
- Answers should be written in your own words
    - Sometimes it is possible to find perfect answers on Wikipedia or Stack Exchange or in good old textbooks.
    - Simply copying the answer of someone else is plagiarism.
    - Copying the answer and providing the reference solves the plagiarism issue but usually does not show that you understood the answer.
    - Hence, we want you to write the answer in your own words.
    - Learning how to write concise and precise answers is an important academic skill.

Culture of Interaction

- I am here to help you learn the material.
- If things are unclear, ask questions.
- If I am going too fast, tell me.
- If I am going too slow, tell me.
- Discussion in class is most welcome - don't be shy.
- Discussion in tutorials is even more welcome - don't be shy.
- If you do not understand something, chances are pretty high your neighbor does not understand either.
- Don't be afraid of asking teaching assistants or myself for help and additional explanations.

Resources

1 Course Objectives and Grading

2 Rules of the Game

3 Resources

Study Material and Forums

- There is no required textbook.
- The slides and notes are available on the course web page.
  `http://cnds.jacobs-university.de/courses/sads-2019`
- We will be using Moodle and it hosts a forum for this course.
  `https://moodle.jacobs-university.de/`
- General questions should be asked on the Moodle forum.
  - Faster responses since many people can answer
  - Better responses since people can collaborate on the answer
- For individual questions, see me at my office (or talk to me after class).

Part: Introduction

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

Motivation

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

Can we trust computers?

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

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.

Classic Computing Disasters

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

IoT Remove Control Light Bulbs 2018

Spectre: Vulnerability of the Year 2018

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

// ...

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

- Is the above code a vulnerability?

Spectre: Main Memory and CPU Memory Caches

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

Spectre: Timing Side Channel Attack

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

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.

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.

Dependability Concepts and Terminology

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

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.

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.

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.

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*.

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 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.

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 Forecasting

Definition (fault forecasting)

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

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

Dependability Metrics

4 Motivation

5 Classic Computing Disasters

6 Dependability Concepts and Terminology

7 Dependability Metrics

Reliability and MTTF/MTBF/MTTR

Definition (reliability)

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

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

Availability

Definition (availability)

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

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

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.

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.

Part: Software Engineering

8 General Aspects

9 Software Testing

10 Software Specification

11 Software Verification

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 Software Development Practices

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

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

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.

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.

Test Coverages

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

Mutation Testing

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

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.

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.

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.

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:

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

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.

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).

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*

Hoare Command Sequences

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

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: $IF\ X < Y\ THEN\ M := Y\ ELSE\ M := X\ FI$

Hoare While Loop

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

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.

Specification can be Tricky

- Specification for the maximum of two variables:

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

- C could be:

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

- But C could also be:

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

- And C could also be:

```
Y := X
```

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

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

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.

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

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

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

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

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.

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:

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

Specification Conjunction and Disjunction

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

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

- We get a similar rule for disjunctions:

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

- These rules allows us to prove $\vdash \{P\}\ C\ \{Q_1 \land Q_2\}$ 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:

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

Conditional Command Rule

- Syntax: $\mathit{IF}\ S\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2\ \mathit{FI}$
- 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:

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

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:

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

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.

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:

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

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

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.

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 $\vdash \{P\}\ C\ \{Q\}$.

Generation of Verification Conditions

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

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.

Rules for Total Correctness [1/4]

- Assignment axiom

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

- Precondition strengthening

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

- Postcondition weakening

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

Rules for Total Correctness [2/4]

- Specification conjunction

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

- Specification disjunction

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

- Skip command rule

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

Rules for Total Correctness [3/4]

- Sequence rule

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

- Conditional rule

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

Rules for Total Correctness [4/4]

- While rule

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

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

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

Generation of Termination Verification Conditions

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

$$P \rightarrow R$$

$$R \wedge \neg S \rightarrow Q$$

$$R \wedge S \rightarrow E \geq 0$$

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

Termination and Correctness

- Partial correctness and termination implies total correctness:

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

- Total correctness implies partial correctness and termination:

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

Part: Concurrency and Distributed Algorithms

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Modeling Concurrent Systems

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Actor Model (Hewitt 1973)

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

Communicating Sequential Processes (Hoare 1978)

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

Logical Clocks (Lamport 1978)

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

π Calculus (Milner 1992)

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

Model of Distributed Algorithms

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Transition System

Definition (transition system)

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

Definition (execution)

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

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

Transition System

Definition (reachability)

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

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

Local Algorithm

Definition (local algorithm)

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

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

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

Distributed Algorithm

Definition (distributed algorithm)

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

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

Induced Async. Transition System

Definition (Induced Async. Transition System)

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

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

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

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

Induced Async. Transition System

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

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

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

for which one of the following three conditions holds:

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

Induced Sync. Transition System

Definition (Induced Sync. Transition System)

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

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

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

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

Induced Sync. Transition System

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

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

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

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

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

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

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

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

Events, Causality, Logical Clocks

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Events and Causal Order

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

Dependence of Events

Theorem

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

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

Causal Order

Definition (causal order)

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

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

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

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

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

Computations

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

Equivalent Executions

Theorem

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

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

Computation

Definition (computation)

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

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

Logical Clocks

Definition (clock)

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

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

Definition (lamport clock)

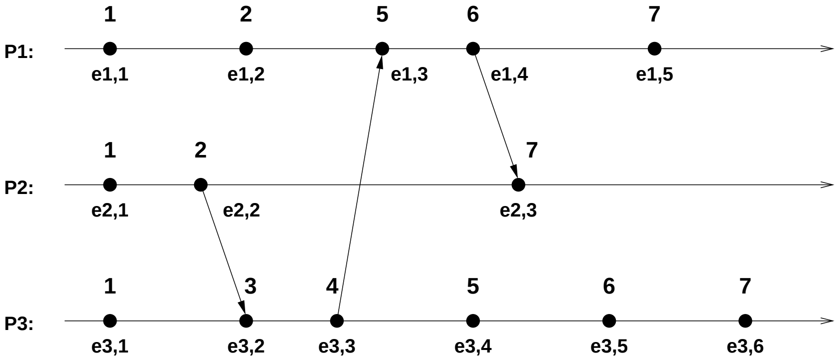
A Lamport clock is a clock function Θ_L which assigns to every event a the length k of the longest sequence $(e_1, \ldots, e_k)$ of events satisfying $e_1 \prec e_2 \prec \ldots \prec e_k = a$.

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

Lamport Clocks

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

Lamport Clock Example



Vector Clocks

Definition (vector clocks)

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

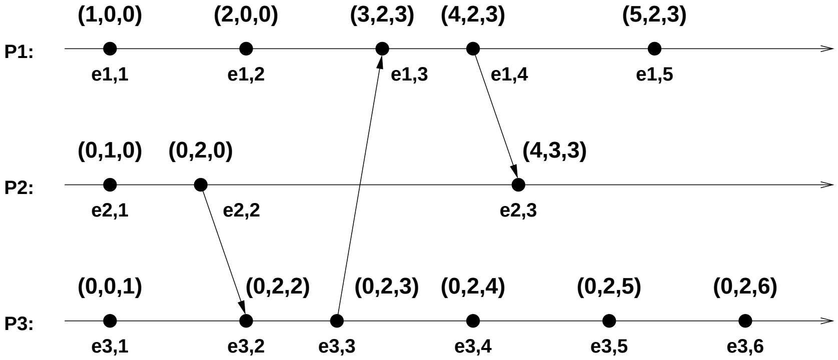
- Vectors are naturally ordered by the vector order:

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

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

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

Vector Clock Example



Stable Properties and Snapshots

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 **Stable Properties and Snapshots**

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Properties of Computations

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

Stable Property Detection Algorithm

```
procedure STABLE-PROPERTY-DETECTION($P$)
    repeat
        $\gamma \leftarrow take\_global\_snapshot()$
        $detected \leftarrow P(\gamma)$
        if $\neg detected$ then
            $suspend\_some\_time()$
        end if
    until $detected$
end procedure
```

Snapshots Preliminaries

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

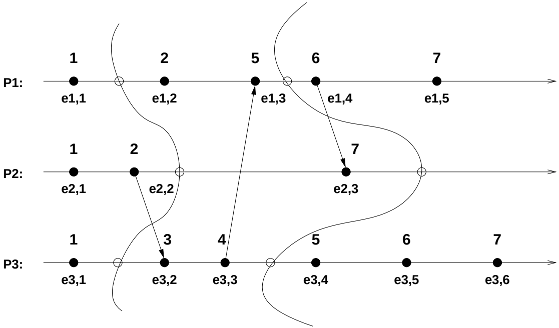
Snapshot Approach (1/2)

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

Snapshot Approach (2/2)

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

Anomalies



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

Feasibile Snapshot and Cuts

Definition (feasible snapshot)

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

Definition (cut)

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

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

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

Consistent Cuts and Meaningful Snapshot

Definition (consistent cut)

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

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

Definition

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

Feasible vs. Meaningful Snapshots vs. Consistent Cuts

Theorem

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

(1) *S^* is feasible.*

(2) *L is a consistent cut.*

(3) *S^* is meaningful.*

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

Chandy-Lamport Algorithm

```
procedure INITIATE
    if $\neg taken_p$ then
        $record\_local\_state()$
        $taken_p \leftarrow true$
        for $\forall q \in Neigh_p$ do
            $send(q, marker)$
        end for
    end if
end procedure
```

```
procedure MARKER-ARRIVED
    $recv(q, marker)$
    if $\neg taken_p$ then
        $record\_local\_state()$
        $taken_p \leftarrow true$
        for $\forall q \in Neigh_p$ do
            $send(q, marker)$
        end for
    end if
end procedure
```

Chandy-Lamport Properties

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

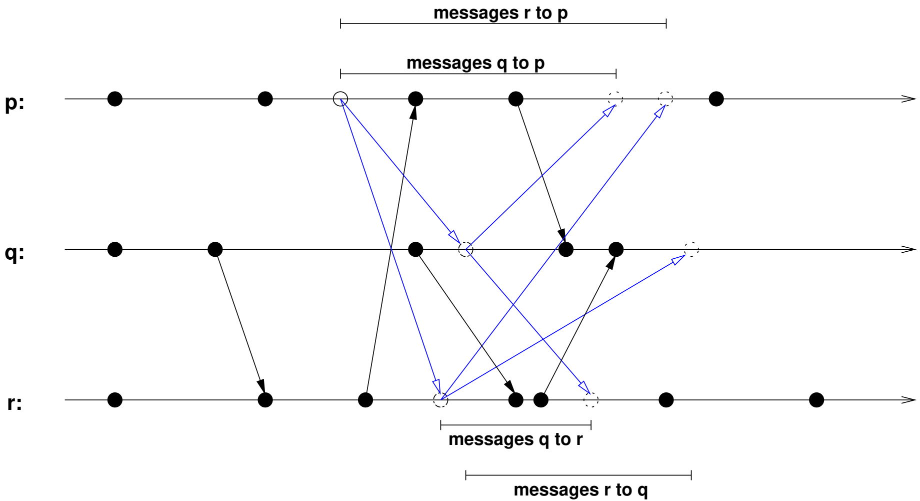
Construction of the Channel State

Lemma

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

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

Chandy-Lamport Snapshot Example



Fault Tolerance and Broadcasts

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

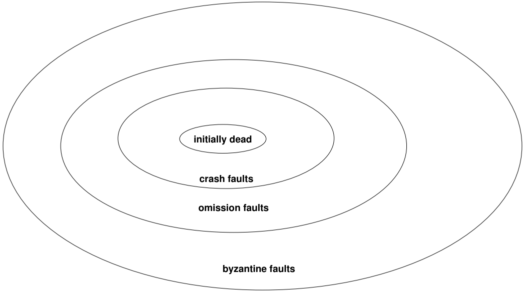
16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

Fault Models

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

Hierarchy of Fault Models



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

Benign vs. Malign Failures

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

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

Approaches to Fault-Tolerance

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

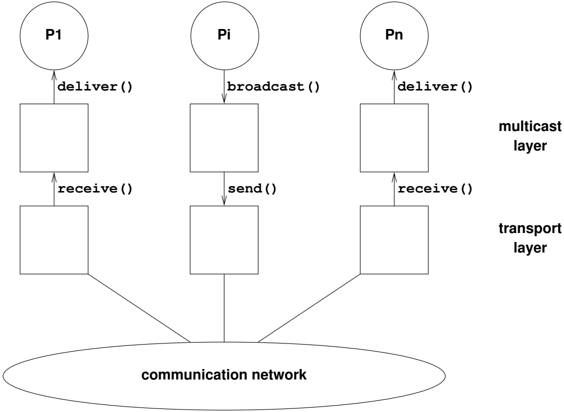
Robust Decision Algorithms

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

Reliable Broadcasts

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

Broadcast System Model



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

Reliable Broadcast

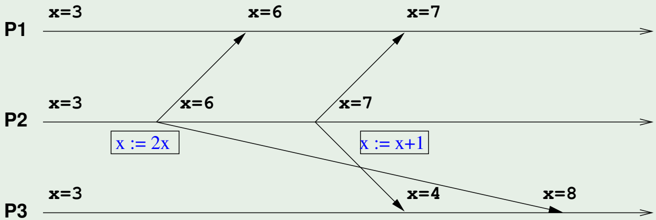
Definition

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

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

FIFO Broadcast

Example

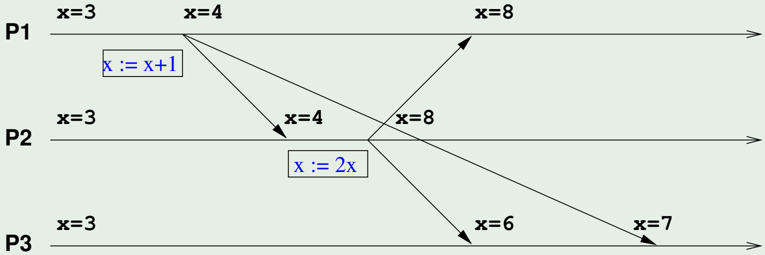


Definition

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

Causal Broadcast

Example

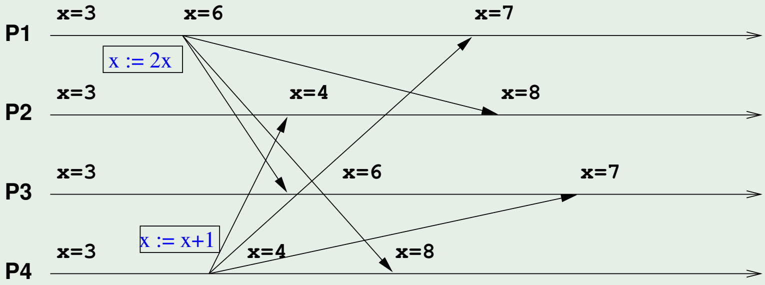


Definition

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

Atomic Broadcast

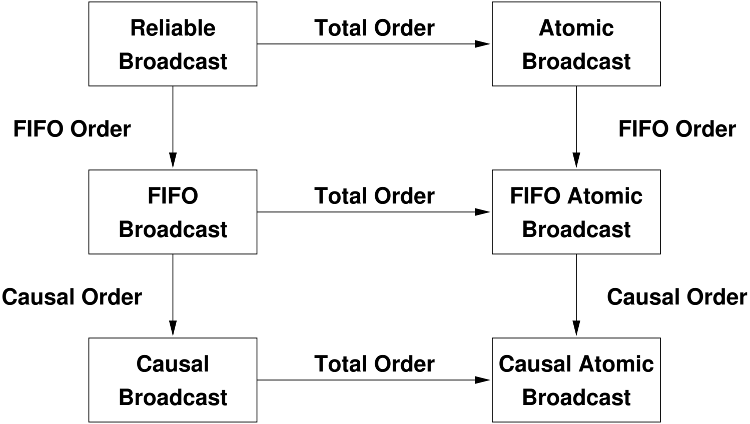
Example



Definition

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

Broadcast Variants



Communicating Sequential Processes

12 Modeling Concurrent Systems

13 Model of Distributed Algorithms

14 Events, Causality, Logical Clocks

15 Stable Properties and Snapshots

16 Fault Tolerance and Broadcasts

17 Communicating Sequential Processes

CSP Notation: Events, Processes, Alphabet, Prefix

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

Recursion and Choice

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

  $$X = F(X)$$

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

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

- If x and y are distinct events, then

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

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

Basic Laws

$$
\begin{align}
STOP &\neq (d \rightarrow P) \tag{L1A}\\
(c \rightarrow P) &\neq (d \rightarrow Q) \quad \text{if } c \neq d \tag{L1B}\\
(c \rightarrow P \mid d \rightarrow Q) &= (d \rightarrow Q \mid c \rightarrow P) \tag{L1C}\\
(c \rightarrow P) = (c \rightarrow Q) &\equiv P = Q \tag{L1D}\\
(Y = F(Y)) &\equiv (Y = \mu X \bullet F(X)) \quad \text{if } F(X) \text{ is a guarded expression} \tag{L2}\\
\mu X \bullet F(X) &= F(\mu X \bullet F(X)) \tag{L2A}
\end{align}
$$

Traces

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

Trace Catenation

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

$$s^\frown\langle\rangle = \langle\rangle^\frown s = s \tag{L1}$$

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

$$s^\frown t = s^\frown u \equiv t = u \tag{L3}$$

$$s^\frown t = u^\frown t \equiv s = u \tag{L4}$$

$$s^\frown t = \langle\rangle \equiv s = \langle\rangle \wedge t = \langle\rangle \tag{L5}$$

$$t^0 = \langle\rangle \tag{L6}$$

$$t^{n+1} = t^\frown t^n \tag{L7}$$

$$t^{n+1} = t^{n\frown} t \tag{L8}$$

$$(s^\frown t)^{n+1} = s^\frown(t^\frown s)^{n\frown} t \tag{L9}$$

Trace Restriction

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

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

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

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

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

Trace Head and Tail

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

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

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

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

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

Trace Star

- The set A^* is the set of all finite traces (including $\langle\rangle$) which are formed from symbols in the set A:

$$A^* = \{s | s \uparrow A = s\}$$

- Basic star laws:

$$\langle\rangle \in A^* \tag{L1}$$

$$\langle x\rangle \in A^* \equiv x \in A \tag{L2}$$

$$(s ^\frown t) \in A^* \equiv s \in A^* \wedge t \in A^* \tag{L3}$$

$$A^* = \{t | t = \langle\rangle \vee (t_0 \in A \wedge t' \in A^*)\} \tag{L4}$$

Trace Ordering

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

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

  and we say that s is a prefix of t.

- Basic ordering laws:

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

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

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

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

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

$$s \leq u \wedge t \leq u \Rightarrow s \leq t \vee t \leq s \tag{L6}$$

Trace Length

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

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

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

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

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

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

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

Traces of a Process

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

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

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

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

Laws of Traces of a Process

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

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

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

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

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

Laws of Traces of a Process

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

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

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

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

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

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

Specification, Satisfaction, Proof

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

Satisfaction and Proof Laws

$$STOP \; sat \; (t = \langle\rangle) \tag{L4A}$$

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

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

$$\begin{aligned} P \; sat \; S(t) \wedge Q \; sat \; T(t) \Rightarrow (c \rightarrow P | d \rightarrow Q) \; sat& \\ (t = \langle\rangle \vee (t_0 = c \wedge S(t')) \vee (t_0 = d \wedge T(t')))& \end{aligned} \tag{L4D}$$

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

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

Interaction of Processes

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

Interaction Laws

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

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

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

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

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

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

$$(x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y)) = (z : (A \cap B) \rightarrow (P(z) \parallel Q(z))) \tag{L4}$$

Concurrency of Processes

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

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

Concurrency Laws

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

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

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

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

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

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

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

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

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

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

Change of Symbols

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

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

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

Labeled Processes

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

Change of Symbols Laws

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

$$f(STOP) = STOP \tag{L1}$$

$$f(x : B \to P(x)) = (y : f(B) \to f(P(f^{-1}(y)))) \tag{L2}$$

$$f(P \parallel Q) = f(P) \parallel f(Q) \tag{L3}$$

$$f(\mu X : \bullet F(X)) = (\mu Y : f(A) \bullet f(F(f^{-1}(Y)))) \tag{L4}$$

$$f(g(P)) = (f \circ g)P \tag{L5}$$

$$traces(f(P)) = \{f^*(s) \mid s \in traces(P)\} \tag{L6}$$

$$f(P)/f^*(s) = f(P/s) \tag{L7}$$

Non-deterministic Choice

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

  $$P \sqcap Q$$

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

Non-deterministic Choice Laws

$$P \sqcap P = P \tag{L1}$$

$$P \sqcap Q = Q \sqcap P \tag{L2}$$

$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \tag{L3}$$

$$x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q) \tag{L4}$$

$$(x : B \rightarrow (P(x) \sqcap Q(x))) = (x : B \rightarrow P(x)) \sqcap (x : B \rightarrow Q(x)) \tag{L5}$$

$$P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R) \tag{L6}$$

$$(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R) \tag{L7}$$

$$f(P \sqcap Q) = f(P) \sqcap f(Q) \tag{L8}$$

General Choice

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

  $$P \,\Box\, Q$$

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

General Choice Laws

$$P \mathbin{\Box} P = P \tag{L1}$$

$$P \mathbin{\Box} Q = Q \mathbin{\Box} P \tag{L2}$$

$$P \mathbin{\Box} (Q \mathbin{\Box} R) = (P \mathbin{\Box} Q) \mathbin{\Box} R \tag{L3}$$

$$P \mathbin{\Box} STOP = P \tag{L4}$$

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

$$P \mathbin{\Box} (Q \sqcap R) = (P \mathbin{\Box} Q) \sqcap (P \mathbin{\Box} R) \tag{L6}$$

$$P \sqcap (Q \mathbin{\Box} R) = (P \sqcap Q) \mathbin{\Box} (P \sqcap R) \tag{L7}$$

Refusals

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

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

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

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

Refusals Laws

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

Concealment

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

Interleaving

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

  $$P \mid\mid\mid Q$$

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

Communication

- A communication event is described by a pair $c.v$ where c is the name of the channel on which the communication takes place and v is the value of the message which passes.
- The set of all messages which a process P can communicate on channel c is defined by:

$$\alpha c(P) = \{v \mid c.v \in \alpha P\}$$

- The functions $\mathit{channel}(c.v) = c$ and $\mathit{message}(c.v) = v$ provide us with the channel c and the message v of the communication event $c.v$.
- A process which writes v to c and then behaves list P is denoted as:

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

- A process which receives x on c and then behaves like $P(x)$ is denoted as:

$$(c?x \rightarrow P(x)) = (y : \{y : \mathit{channel}(y) = c\} \rightarrow P(\mathit{message}(y)))$$

Communication Choice

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

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

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

Communication Laws

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

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

Chaining (Pipes)

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

$$\begin{aligned} \alpha(P \gg Q) &= \alpha \mathit{left}(P) \cup \alpha \mathit{right}(Q) \\ \alpha \mathit{right}(P) &= \alpha \mathit{left}(Q) \end{aligned}$$

Chaining Laws

$$P \gg (Q \gg R) = (P \gg Q) \gg R \tag{L1}$$

$$(\mathit{right}!v \rightarrow P) \gg (\mathit{left}?y \rightarrow Q(y)) = P \gg Q(v) \tag{L2}$$

$$(\mathit{right}!v \rightarrow P) \gg (\mathit{right}!w \rightarrow Q) = \mathit{right}!w \rightarrow ((\mathit{right}!v \rightarrow P) \gg Q) \tag{L3}$$

$$(\mathit{left}?x \rightarrow P(x) \gg (\mathit{left}?y \rightarrow Q(y)) = \mathit{left}?x \rightarrow (P(x) \gg (\mathit{left}?y \rightarrow Q(y))) \tag{L4}$$

$$\begin{aligned}(\mathit{left}?x \rightarrow P(x)) \gg (\mathit{right}!w \rightarrow Q) &= (\mathit{left}?x \rightarrow (P(x) \gg (\mathit{right}!w \rightarrow Q) \\ &\quad \mid \mathit{right}!w \rightarrow ((\mathit{left}?x \rightarrow P(x)) \gg Q))\end{aligned} \tag{L5}$$

$$\begin{aligned}(\mathit{left}?x \rightarrow P(x)) \gg R \gg (\mathit{right}!w \rightarrow Q) &= (\mathit{left}?x \rightarrow (P(x) \gg R \gg (\mathit{right}!w \rightarrow Q) \\ &\quad \mid \mathit{right}!w \rightarrow ((\mathit{left}?x \rightarrow P(x)) \gg R \gg Q))\end{aligned} \tag{L6}$$

$$R \gg (\mathit{right}!w \rightarrow Q) = \mathit{right}!w \rightarrow (R \gg Q) \tag{L7}$$

$$(\mathit{left}?x \rightarrow P(x)) \gg R = \mathit{left}?x \rightarrow (P(x) \gg R) \tag{L8}$$

Part: Cryptography

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Cryptography Primer

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Try to read the following text. . .

```
Jrypbzr gb Frpher naq Qrcraqnoyr Flfgrzf!

W!eslmceotmsey St oe lSbeacdunreep eaDn d

J!rfyzprbgzfrl Fg br yFornpqhaerrc rnQa q
```

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.

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:

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

Cryptosystem Requirements

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

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.

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).

Digital Signatures

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

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)

Symmetric Encryption Algorithms and Block Ciphers

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Substitution Ciphers

Definition (monoalphabetic and polyalphabetic substitution ciphers)

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

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

Permutation Cipher

Definition (permutation cipher)

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

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

Product Cipher

Definition (product cipher)

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

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

Chosen-Plaintext and Chosen-Ciphertext Attack

Definition (chosen plaintext attack)

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

Definition (chosen ciphertext attack)

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

Polynomial and Negligible Functions

Definition (polynomial and negligible functions)

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

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

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.

One-way Functions

Definition (one-way function)

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

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

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 $1/2$.

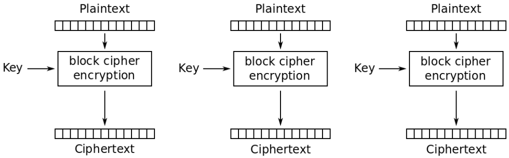
Block Cipher

Definition (block cipher)

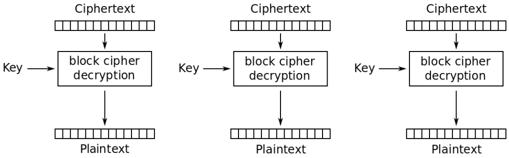
A *block cipher* is a cipher that operates on fixed-length groups of bits called a block.

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

Electronic Code Book Mode

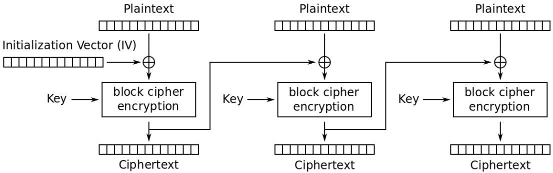


Electronic Codebook (ECB) mode encryption

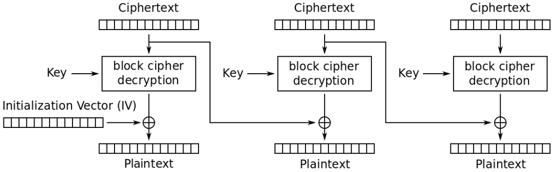


Electronic Codebook (ECB) mode decryption

Cipher Block Chaining Mode

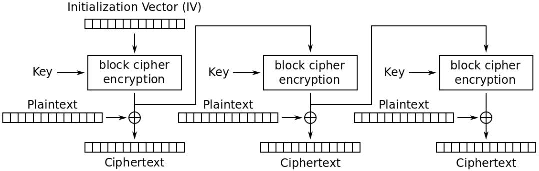


Cipher Block Chaining (CBC) mode encryption

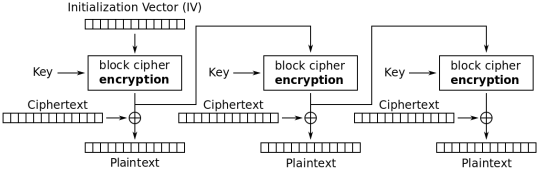


Cipher Block Chaining (CBC) mode decryption

Output Feedback Mode

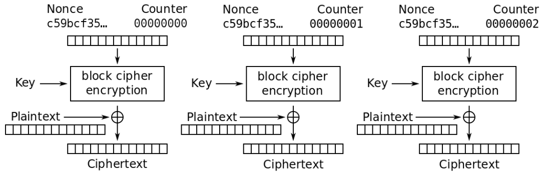


Output Feedback (OFB) mode encryption

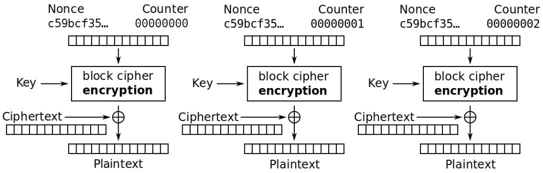

Initialization Vector (IV)
Key
block cipher **encryption**
Key
block cipher **encryption**
Key
block cipher **encryption**
Ciphertext
Ciphertext
Ciphertext
Plaintext
Plaintext
Plaintext

Output Feedback (OFB) mode decryption

Counter Mode



Counter (CTR) mode encryption



Counter (CTR) mode decryption

Substitution-Permutation Networks

Definition (substitution-permutation network)

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

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

Advanced Encryption Standard (AES)

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

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

Asymmetric Encryption Algorithms

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Asymmetric Encryption Algorithms

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

Rivest-Shamir-Adleman (RSA)

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

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.

Cryptographic Hash Functions

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Cryptographic Hash Functions

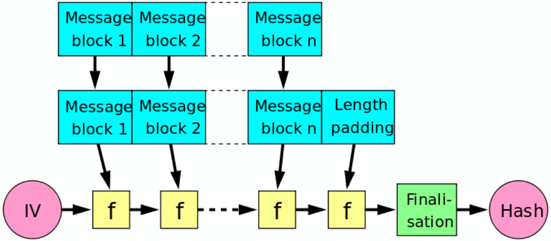
- Cryptographic hash functions serve many purposes:
  - data integrity verification
  - integrity verification and authentication (via keyed hashes)
  - calculation of fingerprints for efficient digital signatures
  - adjustable proof of work mechanisms
- A cryptographic hash function can be obtained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

Cryptographic Hash Functions

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

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

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.

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.

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)

$$E_k(M) \,\|\, H_k(E_k(M))$$

- Encrypt-and-Mac (EaM)

$$E_k(M) \,\|\, H_k(M)$$

- Mac-then-Encrypt (MtE)

$$E_k(M \,\|\, H_k(M))$$

Digital Signatures and Certificates

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Digital Signatures

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

Digital Signatures using Asymmetric Cryptosystems

- Direct signature of a document m:
  - Signer: $S = E_{k^{-1}}(m)$
  - Verifier: $D_k(S) \overset{?}{=} m$
- 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.

Public Key Certificates

Definition (public key certificate)

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

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

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

Public Key Infrastructure (PKI)

Definition

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

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

X.509 Certificate ASN.1 Definition

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

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

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
     }
```

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 }
```

Key Exchange Schemes

18 Cryptography Primer

19 Symmetric Encryption Algorithms and Block Ciphers

20 Asymmetric Encryption Algorithms

21 Cryptographic Hash Functions

22 Digital Signatures and Certificates

23 Key Exchange Schemes

Cryptographic Protocol Notation

$A, B, \ldots$	principals
$K_{AB}, \ldots$	symmetric key shared between A and B
$K_A, \ldots$	public key of A
$K_A^{-1}, \ldots$	private key of A
H	cryptographic hash function
$N_A, N_B, \ldots$	nonces (fresh random messages) chosen by A, B, $\ldots$
P, Q, R	variables ranging over principals
X, Y	variables ranging over statements
K	variable over a key
$\{m\}_K$	message m encrypted with key K

Key Exchange and Ephemeral Keys

Definition (key exchange)

Key exchange (also key establishment) is any method by which cryptographic keys are exchanged between two parties, allowing use of a cryptographic algorithm.

- Key exchange methods are important to establish ephemeral keys even if two principals have already access to suitable long-term keys
- Ephemeral keys help to protect keys that are used to bootstrap secure communication between principals
- Ephemeral keys can provide perfect forward secrecy

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:

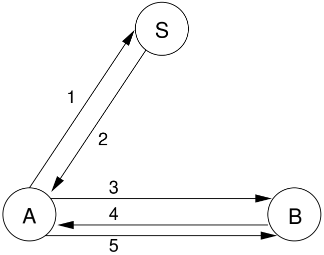
$$K_{AB} = y_A^{x_B} \bmod p = (g^{x_A} \bmod p)^{x_B} \bmod p = g^{x_A x_B} \bmod p$$

  - A and B now own a shared key K_{AB}.

Diffie-Hellman Key Exchange (cont.)

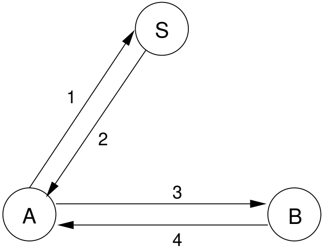
- A number g is a primitive root of $\mathbb{Z}_p = \{0, \ldots, p-1\}$ if the sequence $g^1 \bmod p, g^2 \bmod p, \ldots, g^{p-1} \bmod p$ produces the numbers $1, \ldots, p-1$ in any permutation.
- p should be choosen such that $(p-1)/2$ is prime as well.
- p should have a length of at least 2048 bits.
- Diffie-Hellman is not perfect: An attacker can play "man in the middle" (MIM) by claiming B's identity to A and A's identity to B.

Needham-Schroeder Protocol



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

Kerberos Protocol



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

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.

Using BAN Logic

- Steps to use BAN logic:
  1. Idealize the protocol in the language of the formal BAN logic.
  2. Define your initial security assumptions in the language of BAN logic.
  3. Use the productions and rules of the logic to deduce new predicates.
  4. Interpret the statements you've proved by this process. Have you reached your goals?
  5. Trim unnecessary fat from the protocol, and repeat (optional).
- BAN logic does not prove correctness of the protocol; but it helps to find subtle errors.

Part: Secure Communication Protocols

24 Pretty Good Privacy

25 Transport Layer Security

26 Secure Shell

Pretty Good Privacy

24 Pretty Good Privacy

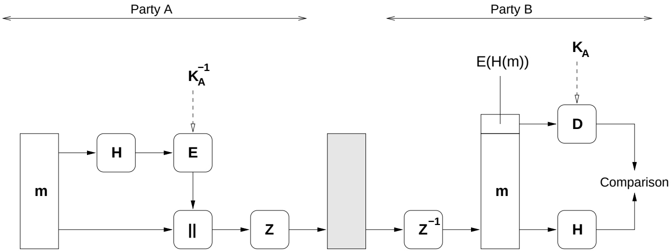
25 Transport Layer Security

26 Secure Shell

Pretty Good Privacy (PGP)

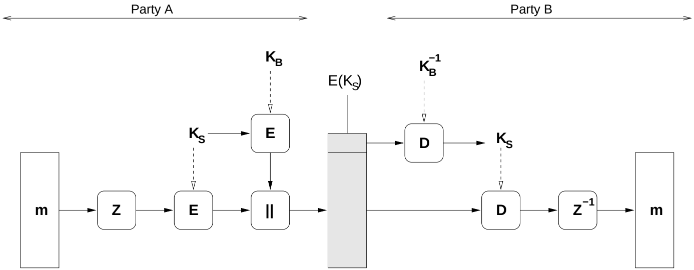
- PGP was developed by Philip Zimmerman in 1991 and it is rather famous because PGP also demonstrated why patent laws and export laws in a globalized world need new interpretations.
- There are nowadays several independent PGP implementations.
- The underlying PGP specification is now called open PGP (RFC 4880).
- A competitor to PGP is S/MIME (which relies on X.509 certificates).

PGP Signatures



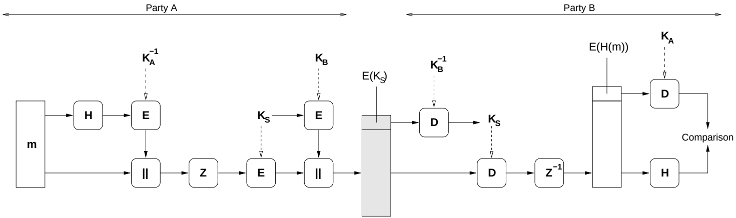
- A computes $c = Z(E_{K_A^{-1}}(H(m))||m)$
- B computes $Z^{-1}(c)$, splits the message and checks the signature by computing $D_{K_A}(E_{K_A^{-1}}(H(m)))$ and then checking the hash $H(m)$.

PGP Confidentiality



- *A* encrypts the message using the key K_s generated by the sender and appended to the encrypted message.
- The key K_s is protected by encrypting it with the public key K_B.
- Symmetric encryption is fast while public-key algorithms make it easier to exchange keys.

PGP Signatures and Confidentiality



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

PGP Key Management

- Keys are maintained in so called key rings (one for public keys and one for private keys).
- Key generation utilizes various sources of random information (`/dev/random` if available) and symmetric encryption algorithms to generate good key material.
- So called “key signing parties” are used to sign keys of others and to establish a “web of trust” in order to avoid centralized certification authorities.

PGP Private Key Ring

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

- Private keys are encrypted using $E_{H(P_i)}()$, which is a symmetric encryption function using a key which is derived from a hash value computed over a user supplied passphrase P_i.
- The Key ID is taken from the last 64 bits of the key K_i.

PGP Public Key Ring

Timestamp	Key ID	Public Key	Owner Trust	User ID	Signatures	Sig. Trust(s)
⋮	⋮	⋮	⋮	⋮	⋮	⋮
T_i	$K_i \bmod 2^{64}$	K_i	otrust_i	User_i	...	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮

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

Transport Layer Security

24 Pretty Good Privacy

25 Transport Layer Security

26 Secure Shell

Transport Layer Security

- Transport Layer Security (TLS), formerly known as Secure Socket Layer (SSL), was created by Netscape to secure data transfers on the Web (i.e., to enable commerce on the Web)
- As a user-space implementation, TLS can be shipped as part of applications (Web browsers) and does not require operating system support
- TLS uses X.509 certificates to authenticate servers and clients (although TLS layer client authentication is not often used)
- TLS is widely used to secure application protocols running over TCP (e.g., http, smtp, ftp, telnet, imap, . . .)
- A datagram version of TLS called DTLS can be used with protocols running over UDP

History of TLS and SSL

Name	Organization	Published	Wire Version
SSL 1.0	Netscape	unpublished	1.0
SSL 2.0	Netscape	1995	2.0
SSL 3.0	Netscape	1996	3.0
TLS 1.0	IETF	1999	3.1
TLS 1.1	IETF	2006	3.2
TLS 1.2	IETF	2008	3.3
TLS 1.3	IETF	2018	3.3 + supported_versions

- TLS 1.3 is brand new, this material follows TLS 1.2 and TLS 1.3

TLS Protocols

- The *Handshake Protocol* authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material.
- The *Alert Protocol* communicates alerts such as closure alerts and error alerts.
- The *Record Protocol* uses the parameters established by the handshake protocol to protect traffic between the communicating peers.
- The Record Protocol is the lowest internal layer of TLS and it carries the handshake and alert protocol messages as well as application data.

TLS Record Protocol

Record Protocol

The record protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, adds a message authentication code, and encrypts and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients.

- The record layer is used by the handshake protocol, the change cipher spec protocol (only TLS 1.2), the alert protocol, and the application data protocol.
- The fragmentation and reassembly provided does not preserve application message boundaries.

TLS 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.

TLS Change Cipher Spec Protocol

Change Cipher Spec Protocol

The change cipher spec protocol is used to signal transitions in ciphering strategies.

- The protocol consists of a single ChangeCipherSpec message.
- This message is sent by both the client and the server to notify the receiving party that subsequent records will be protected under the newly negotiated CipherSpec and keys.
- This protocol does not exist anymore in TLS 1.3.

TLS Alert Protocol

Alert Protocol

The alert protocol is used to signal exceptions (warnings, errors) that occured during the processing of TLS protocol messages.

- The alert protocol is used to properly close a TLS connection by exchanging `close_notify` alert messages.
- The closure exchange allows to detect truncation attacks.

Secure Shell

24 Pretty Good Privacy

25 Transport Layer Security

26 Secure Shell

Secure Shell (SSH)

- SSH provides a secure connection through which user authentication and several inner protocols can be run.
- The general architecture of SSH is defined in RFC 4251.
- SSH was initially developed by Tatu Ylonen at the Helsinki University of Technology in 1995, who later founded SSH Communications Security.
- SSH was quickly adopted as a replacement for insecure remote login protocols such as telnet or rlogin/rsh.
- Several commercial and open source implementations are available running on almost all platforms.
- SSH is a Proposed Standard protocol of the IETF since 2006.

SSH Protocol Layers

1. The **Transport Layer Protocol** provides server authentication, confidentiality, and integrity with perfect forward secrecy
2. The **User Authentication Protocol** authenticates the client-side user to the server
3. The **Connection Protocol** multiplexes the encrypted data stream into several logical channels

⇒ SSH authentication is not symmetric!

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

SSH Keys, Passwords, and Passphrases

Host Key

Every machine must have a public/private host key pair. Host Keys are often identified by their fingerprint.

User Key

Users may have their own public/private key pairs.

User Password

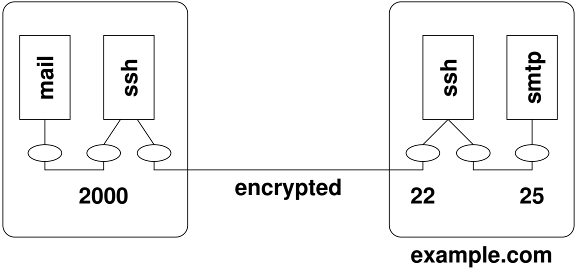
Accounts may have passwords to authenticate users.

Passphrase

The storage of a user's private key may be protected by a passphrase.

SSH Features: TCP Forwarding

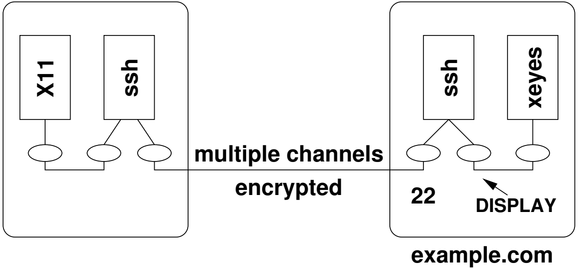
```
ssh -f joe@example.com -L 2000:example.com:25 -N
```



- TCP forwarding allows users to tunnel unencrypted traffic through an encrypted SSH connection.

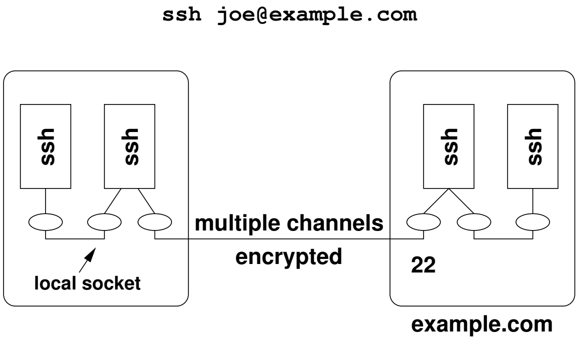
SSH Features: X11 Forwarding

```
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).

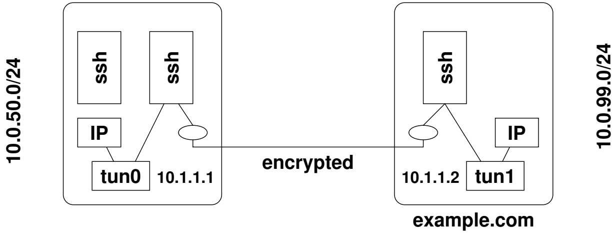
SSH Features: Connection Sharing



- New SSH connections hook as a new channel into an existing SSH connection, reducing session startup times (speeding up shell features such as tab expansion).

SSH Features: IP Tunneling

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

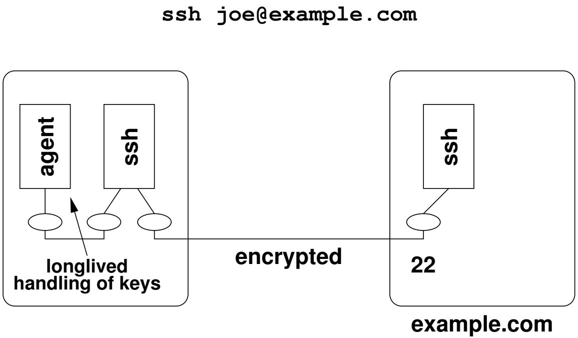


```
ifconfig tun0 10.1.1.1 10.1.1.2 \
          netmask 255.255.255.255
route add 10.0.99.0/24 10.1.1.2
```

```
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
```

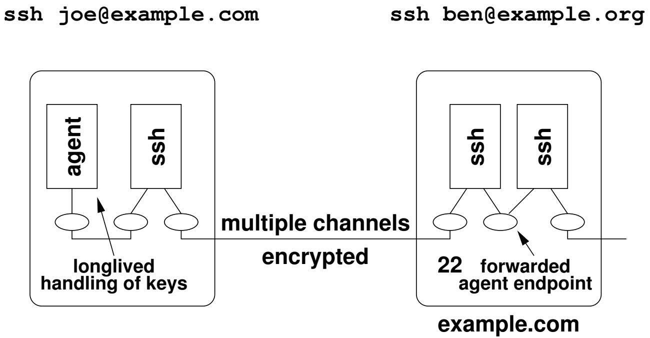
- Tunnel IP packets over an SSH connection by inserting tunnel interfaces into the kernels and by configuring IP forwarding.

SSH Features: SSH Agent



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

SSH Features: SSH Agent Forwarding



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

SSH Transport Protocol

- Transport Protocol (RFC 4253) provides
  - strong encryption,
  - server authentication,
  - integrity protection, and
  - optionally compression.
- SSH transport protocol typically runs over TCP
- 3DES (required), AES128 (recommended)
- hmac-sha2-256 (recommended, see RFC 6668)
- Automatic key re-exchange, usually after 1 GB of data have been transferred or after 1 hour has passed, whichever is sooner.

SSH Key Exchange

- The SSH host key exchange identifies a server by its hostname or IP address and possibly port number.
- Other key exchange mechanisms use different naming schemes for a host.
- Different key exchange algorithms
  - Diffie-Hellman style key exchange
  - GSS-API style key exchange
- Different Host key algorithms
  - Host key used to authenticate key exchange
  - SSH RSA and DSA keys
  - X.509 (under development)

SSH User Authentication

- Executes after transport protocol initialization (key exchange) to authenticate client.
- Authentication methods:
  - Password (classic password authentication)
  - Interactive (challenge response authentication)
  - Host-based (uses host key for user authentication)
  - Public key (usually DSA or RSA keypairs)
  - GSS-API (Kerberos / NETLM authentication)
  - X.509 (under development)
- Authentication is client-driven.

SSH Connection Protocol

- Allows opening of multiple independent channels.
- Channels may be multiplexed in a single SSH connection.
- Channel requests are used to relay out-of-band channel specific data (e.g., window resizing information).
- Channels commonly used for TCP forwarding.

OpenSSH Privilege Separation

- Privilege separation is a technique in which a program is divided into parts which are limited to the specific privileges they require in order to perform a specific task.
- OpenSSH is using two processes: one running with special privileges and one running under normal user privileges
- The process with special privileges carries out all operations requiring special permissions.
- The process with normal user privileges performs the bulk of the computation not requiring special rights.
- Bugs in the code running with normal user privileges do not give special access rights to an attacker.

Part: Information Hiding and Privacy

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

Steganography and Watermarks

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

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

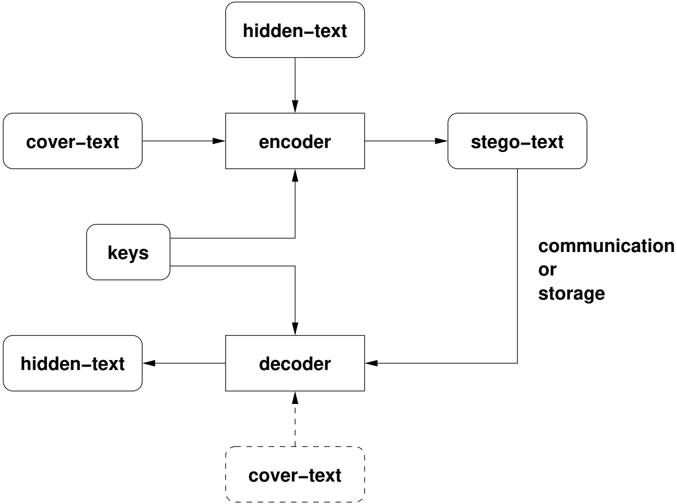
Steganography

Definition (steganography)

Steganography is the embedding of some information (hidden-text) within digital media (cover-text) so that the resulting digital media (stego-text) looks unchanged (imperceptible) to a human/machine.

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

Steganography Workflow



Types of Cover Media

- Information can be hidden in various cover media types:
  - Image files
  - Audio files
  - Video files
  - Text files
  - Software (e.g., executable files, source code)
  - Network traffic (e.g., covert channels)
  - Storage devices (e.g., steganographic file systems)
  - Events (e.g., timing covert channels, signaling covert channels)
  - ...
- Media types of large size usually make it easier to hide information.
- Robust steganographic methods may survive some typical modifications of stego-texts (e.g., cropping or recoding of images).

Watermarking

Definition (watermarking)

Watermarking is the embedding some information (watermark) within a digital media (cover-text) so that the resulting digital media looks unchanged (imperceptible) to a human/machine.

- Watermarking:
  - The hidden information itself is not important.
  - The watermark says something about the cover-text.
- Steganography:
  - The cover-text is not important, it only conveys the hidden information.
  - The hidden text is the valuable information, and it is independent of cover-text.

Classification of Steganographic Algorithms

- fragile vs. robust
  - Fragile: Modifiations of stego-text likly destroys hidden text.
  - Robust: Hidden text is likely to survive modifications of the stego-text.
- blind vs. semi-blind vs. non-blind
  - Blind requires the original cover-text for detection / extraction.
  - Semi-blind needs some information from the embedding but not the whole cover-text
  - Non-blind does not need any information for detection / extraction.
- pure vs. symmetric (secret key) vs. asymmetric (public key)
  - Pure needs no key for detection / extraction.
  - Secret key needs a symmetric key for embedding and extraction.
  - Public key needs a secret key for embedding and a public key for extraction.

LSB-based Image Steganography

- Idea:
  - Some image formats encode a pixel using three 8-bit color values (red, green, blue).
  - Changes in the least-significant bits (LSB) are difficult for humans to see.
- Approach:
  - Use a key to select some least-significant bits of an image to embed hidden information.
  - Encode the information multiple times to achieve some robustness against noise.
- Problem:
  - Existence of hidden information may be revealed if the statistical properties of least-significant bits change.
  - Fragile against noise such as compression, resizing, cropping, rotating or simply additive white Gaussian noise.

DCT-based Image Steganography

- Idea:
  - Image formats such as JPEG use discrete cosine transforms (DCT) to encode image data.
  - The manipulation happens in the frequency domain instead of the spatial domain and this reduces visual attacks against the JPEG image format.
- Approach:
  - Replace the least-significant bits of some of the discrete cosine transform coefficients.
  - Use a key to select some DCT coefficients of an image to embed hidden information.
- Problem:
  - Existence of hidden information may be revealed if the statistical properties of the DCT coefficients are changed.
  - This risk may be reduced by using an pseudo-random number generator to select coefficients.

Covert Channels

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

Covert Channels

- Covert channels represent unforeseen communication methods that break security policies. Network covert channels transfer information through networks in ways that hide the fact that communication takes place (hidden information transfer).
- Covert channels embed information in
  - header fields of protocol data units (protocol messages)
  - the timing of protocol data units (e.g., inter-arrival times)
- We are not considering here covert channels that are constructed by exchanging steganographic objects in application messages.

Covert Channel Patterns

P1 Size Modulation Pattern
The covert channel uses the size of a header field or of a protocol message to encode hidden information.

P2 Sequence Pattern
The covert channel alters the sequence of header fields to encode hidden information.

P3 Add Redundancy Pattern
The covert channel creates new space within a given header field or within a message to carry hidden information.

P4 PDU Corruption/Loss Pattern
The covert channel generates corrupted protocol messages that contain hidden data or it actively utilizes packet loss to signal hidden information.

Covert Channel Patterns

P5 Random Value Pattern
The covert channel embeds hidden data in a header field containing a “random” value.

P6 Value Modulation Pattern
The covert channel selects one of values a header field can contain to encode a hidden message.

P7 Reserved/Unused Pattern
The covert channel encodes hidden data into a reserved or unused header field.

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

Covert Channel Patterns

P9 Rate Pattern
The covert channel sender alters the data rate of a traffic flow from itself or a third party to the covert channel receiver.

P10 Protocol Message Order Pattern
The covert channel encodes data using a synthetic protocol message order for a given number of protocol messages flowing between covert sender and receiver.

P11 Re-Transmission Pattern
A covert channel re-transmits previously sent or received protocol messages.

Anonymization Terminology

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

Anonymity

Definition (anonymity)

Anonymity of a subject from an attacker's perspective means that the attacker cannot sufficiently identify the subject within a set of subjects, the anonymity set.

- All other things being equal, anonymity is the stronger, the larger the respective anonymity set is and the more evenly distributed the sending or receiving, respectively, of the subjects within that set is.
- Robustness of anonymity characterizes how stable the quantity of anonymity is against changes in the particular setting, e.g., a stronger attacker or different probability distributions.

Unlinkability and Linkability

Definition (unlinkability)

Unlinkability of two or more items of interest (IOIs) (e.g., subjects, messages, actions, . . .) from an attacker's perspective means that within the system, the attacker cannot sufficiently distinguish whether these IOIs are related or not.

Definition (linkability)

Linkability of two or more items of interest (IOIs) (e.g., subjects, messages, actions, . . .) from an attacker's perspective means that within the system, the attacker can sufficiently distinguish whether these IOIs are related or not.

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.

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

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.

Identifiability and Identity

Definition (identifiability)

Identifiability of a subject from an attacker's perspective means that the attacker can sufficiently identify the subject within a set of subjects, the identifiability set.

Definition (identity)

An identity is any subset of attribute values of an individual person which sufficiently identifies this individual person within any set of persons. So usually there is no such thing as "the identity", but several of them.

Identity 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.

Mixes and Onion Routing

27 Steganography and Watermarks

28 Covert Channels

29 Anonymization Terminology

30 Mixes and Onion Routing

Mix Networks

- A mix network uses special proxies called mixes to send date from a source to a destination.
- The mixes filter, collect, recode, and reorder messages in order to hide conversations. Basic operations of a mix:
    1. Removal of duplicate messages (an attacker may inject duplicate message to infer something about a mix.
    2. Collection of messages in order to create an ideally large anonymity set.
    3. Recoding of messages so that incoming and outgoing messages cannot be linked.
    4. Reordering of messages so that order information cannot be used to link incoming and outgoing messages.
    5. Padding of messages so that message sizes do not reveal information to link incoming and outgoing messages.

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.

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.

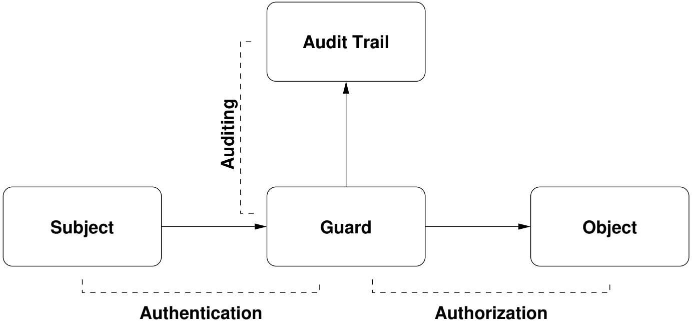
Part: Operating System Security

31 Access Control Principles

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

Lampson Model



- This basic model works well for modeling static access control systems.
- Dynamic access control systems allowing dynamic changes to the access control policy are difficult to model with this approach.

Authentication

31 Access Control Principles

Access Control

31 Access Control Principles

Subjects, Objects, Rights

- Subjects (S): set of active objects
  - processes, users, ...
- Objects (O): set of protected entities
  - files, directories, ...
  - memory, devices, sockets, ...
  - processes, memory, ...
- Rights (R): set of operations a subject can perform on an object
  - create, read, write, delete ...
  - execute ...

Lampson's Access Control Matrix

- Subjects $s_i \in S$ are row headings.
- Objects $o_j \in O$ are column headings.
- The rights of subject s_i when accessing object o_j are defined by the entry $M[s_i, o_j]$ of the access control matrix M
- Another way to look at access control rights is that the rights $r \in R$ are a function mapping $(S \times O) \to R^*$
- 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.

Discretionary vs. Mandatory Access Control

- Discretionary Access Control (DAC)
  - Subjects (with certain permissions, i.e., ownership) can define access control rules to allow or deny access to an object
  - It is the at the subject's discretionary what to allow to whom.
- Mandatory Access Control (MAC)
  - System mechanisms control access to an object and an individual subject cannot alter the access rights.
  - What is allowed is mandated by the system (or the security administrator of the system).

Access Control Lists

- An access control list represents a column of the access control matrix.
- Given a set of subjects S and a set of rights R, an access control list of an object $o \in O$ is a set of tuples of $S \times R^*$.
- Example: Unix file system permissions. The inode of a file (the object) stores the whether a user or a group (the subject) has read/write/execute permissions (the rights).

Access Control List Design Issues

- Who can define and modify ACLs?
- Does the ACL support groups or wildcards?
- How are contradictory ACLs handled?
- Is there support for default ACLs?

Capabilities

- A capability represents a row of the access control matrix.
- Given a set of objects o and a set of rights R, a capability of a subject s is a set of tuples of $O \times R^*$.
- Example: An open Unix file descriptor can be seen as a capability. Once opened, the file can be used regardless whether the file is deleted or whether access rights are changed. The capability (the open file descriptor) can be transfered to child processes. Note that passing capabilities to child processes is not meaningful for all capabilities.

Capabilities Design Issues

- How are capabilities stored?
- How are capabilities protected?
- Can capabilities be passed on to other subjects?
- Can capabilities be revoked?

Access Control Lists vs. Capabilities

- Both are theoretically equivalent (since both at the end can represent the same access control matrix)
- Capabilities tend to be more efficient if the common question is “Given a subject, what objects can it access and how?”.
- Access control lists tend to be more efficient if the common question is “Given an object, what subjects can access it and how?”.
- Access control lists tend to be more popular because they are more efficient when an authorization decision needs to be made.
- Systems often use a mixture of both approaches.