Cloud Computing Architecture Summary

Adrian Hirt

June 2022

1 Performance Analysis

1.1 Interactive Law

- Think time Z
- Response time R
- Number of clients N
- Total cycle time $R + Z$
- Throughput $X = \frac{Jobs}{Time} = \frac{N \cdot \frac{T}{R+Z}}{T} = \frac{N}{R+Z}$
- Response time $R = \frac{N}{X} - Z$

2 Queuing Theory

2.1 Some formulas

- Mean interarrival time: $E[\tau]$
- Mean arrival rate: $\lambda = 1/E[\tau]$
- Service time per job: s
- Mean service rate: $\mu = 1/E[s]$
- Total time: T
- Busy time: B
- Utilization: $p = B/T$
- Jobs completed: C
- Throughput: $X = C/T = C/B * B/T = p * \mu$
- Utilization in open system: $p = \lambda/\mu = \lambda * E[s]$

2.2 Open System

In an open system, if $\lambda > \mu$, the system is unstable and the queue will grow without bounds.
This means the throughput will remain the service rate μ, but the response time will go to infinity.
In an open system, we have that the throughput grows with arrival rate λ until it matches the service rate μ, after which the system becomes unstable.

2.3 Little's Law

- $n = n_s + n_q$, where
  - n is the number of jobs in the system
  - n_s is the number of jobs in the service
  - n_q is the number of jobs in the queue
- $w = w_q + s$, where
  - w is the total time in the system
  - w_q is the time waiting
  - s is the time in the service

2.3.1 Little's law for open systems

- Queuing System: $E[n] = \lambda \cdot E[w]$
- Queue: $E[n_q] = \lambda \cdot E[w_q]$
- $E[n] = E[n_q] + E[n_s]$
- $E[w] = E[w_q] + E[s] = E[w_q] + 1/\mu$

2.3.2 Little's law for closed systems

Similar to open systems but with $N = X \cdot (R + Z)$

2.4 Basic models

2.4.1 M/M/1

Represents a system having one server with infinite buffers, using FIFO queues. Mean arrival rate of λ and mean service rate of μ.

- Utilization (probab. of 1 or more jobs in system): $p = \lambda/\mu$
- Number of jobs: $E[n] = \frac{p}{1-p}$
- Response time: $E[w] = \frac{1/\mu}{1-p}$

2.4.2 M/M/m

Represents a system with m servers, where each server can serve μ jobs per unit of time.
If less than m jobs in the system, the jobs are served right away.

3 Virtualization

3.1 Main requirements

The three main requirements for virtualization are:

1. **Safety**: Isolation between guests, isolation between guest and VMM
2. **Equivalency**: Results shold be the same as if the programs would run on bare metal.
3. **Efficiency**: Good performance, minimal overhead

3.2 Approaches to virtualization

Hosted interpretation
The first approach is using **hosted interpretation**, which means running the VMM as a regular user application on top of a host OS. The VMM maintains a software-representation of the hardware.
Pros:

- Complete isolation
- Easy to handle privileged instructions

Cons:

- Emulating a modern processor is difficult
- Very slow

Direct execution with trap-and-emulate
Here, we run the VMM directly on the host hardware. Whenever the guest executes a privileged instruction, it results in a trap which is handled by the VMM using a policy.
Pros:

- Faster than using hosted interpretation

Cons:

- Doesn't always work (processor needs to be virtualizable, see next section).
- Still relatively slow

Binary translation
This approach involves dynamically rewriting the instruction stream to enable the VMM to intercept sensitive instructions which do not trap. These instructions are rewritten such that they trap.
Pros:

- Can run unmodified guest OS and apps
- Most instructions run at bare-metal speed

Cons:

- Implementing VMM is difficult
- Translation impacts performance

Paravirtualization
This approach involves modifying the guest OS to trap on sensitive instructions and/or provide another virtualization support at the OS software level, for example by modifying sensitive-but-unprivileged instructions.
Pros:

- Faster than direct execution with translation

Cons:

- Requires substantial modifications to the guest OS

Direct execution with hardware support
This approach adds hardware support for virtualization to the CPU. For this, we add a new privilege mode **VT root mode**, which allows direct execution of VM on the processor (*vmentry*), until a privileged instruction is executed (*vmexit*).
It also adds a **virtual machine control structure [VMCS]**, which can be configured to control whih instructions trigger a *vmexit*.
The CPU can now operate in two modes:
In **root mode**, the CPU operates like older CPUs with 4 privilege levels (ring 0-3). The hypervisor runs in root mode and can configure the VMCS to cause a *vmexit* (hardware transfer of control to VMM) when a sensitive instruction is executed.
In **non-root mode** we still have 4 privilege levels and the same instruction set, but the instructions are "supervised" by the VMCS.
Pros:

- Fast and supported on most CPUs today. The most common approach today.

Cons:

- VMM must provide illusion of contiguous, zero-based memory.
- VMM needs to intercept paging operations
- *vmexit* add to execution time.

3.3 Virtualizable processor architectures

For a processor architecture to be considered *virtualizable*, two conditions must be satisfied:

1. The processor must support at least two privilege modes (user and kernel).
2. All sensitive instruction must trap (i.e. sensitive instructions must be privileged instructions).

If we have an architecture which is not virtualizable, we can implement virtualization using either **binary translation** or **paravirtualization**.

3.4 Serverless computing

3.4.1 Cold vs warm start

A **cold start** describes the case where an execution unit (e.g. container or VM) for the function which we want to execute needs to be loaded an started from scratch. A **warm start** is the case where the execution unit is already loaded into memory and ready for the function invocation.

As serverless functions are usually very short-lived (ranging from miliseconds to a few minutes) and billed as a microsecond granularity, the time it takes for the execution unit to start-up needs to be short, as this can be a significant percentage of the end-to-end latency.

3.5 Linux mechanisms

- **chroot** changes the root directory of the current process and its children, which means they cannot acces files outside the designate directory tree.
- **namespaces** are used to partition the resources in a system, which can be used to control what set of resources a process can see.
- **cgroups** is a kernel-feature that allows fine-grained allocation & monitoring of resources.
- **seccomp-bpf** is a kernel feature which allows a process to renounce a set if system calls, which is a security feature with the idea to never give a process more privileges than it requires.

4 Cluster management

Cluster management is mainly concerned with **resource allocation** and **resource assignment**. Resource allocation is concerned with how much CPU, DRAM, disk, network bandwidth and other resources are given to an application. Resource assignment on the other hand is concerned with which nodes these resources should be allocated on.

4.1 Resource allocation

We can either use **private resource allocation** or **shared resource allocation** to assign resources.

4.1.1 Private resource allocation

Here, each app receives a private, static set of resources, which is also known as static partitioning.
This approach is simple, we have performance isolation (high load on one app does not affect another app in any way) and it allows for hardware specialization. However, we also have low utilization and failures & maintenance are more difficult to handle.

4.1.2 Shared resource allocation

Here we share machine resources between applications. We need a **scheduler** to ensure:

- **Fairness**: users should be granted the resources they are paying for, and we need some flexibility to express priorities. **Efficient resource usage**: Resources should not be idling while there are users whose demand is not fully satisfied. **Isolation**

4.1.3 Max-min fair share

To share a single resource, e.g. CPU cores, we can use fair sharing, or its generalized version **max-min fair share**. This can be achieved as follows:

- Allocate resources in order or increasing demand
- No source gets a resource larger than its demand
- At max. sources get an equal share of the resource

Max-min fairness is powerful, as it has **share guarantee** (each user gets at least $\frac{1}{n}$ of the resouce, but less if the demand is less) and it's also **strategy-proof**, as the users are not better off by asking for more resources than they need.

However, as a drawback, max-min fair share only works for a single resource, however tasks consume CPU, memory, netword and disk I/O.

4.1.4 Dominant resource fairness (DRF)

With DRF, we apply max-min fairness to dominant shares of resources. For this, we identify the dominant resouce share of each user, and maximize the minumum dominant share across all users.
Here, an users **dominant resource** is the resource that the user has the biggest share of. The **dominant share** is the fraction of the dominant resource allocated.

4.1.5 Token bucket

The max-min fairness approach has a few drawbacks, mainly that an users allocation depends on the demands of other users sharing the resources. Especially with bursts of usage (e.g. in network traffic), the max-min fairness approach does not work that well.

For this, we use the **token bucket** approach, which guarantees a baseline usage, but also allows for bursts. The idea is to control usage by "delaying" it until enough tokens are accumulated. This approach is commonly used for network and storage traffic.

We have a *bucket* with a capacity B, and tokens are added at a rate of r (usually proportional to how much the user is paying). Overflow tokens are dropped, i.e. we don't have more than B tokens in the bucket. When a request arrives, we check if we have sufficient tokens, if yes we forward the requests, else we drop or queue the requests.

4.2 Resouce assignment

To decide which node we pick for a VM / container, we first filter machines that satisfy hard constraints (e.g. if the VM needs a machine with a GPU), and then we rank the canditate nodes to find the machine that best satisfies soft constraints.

5 Systems for machine learning

5.1 Types of ML parallelism

5.1.1 Data parallelism

Here, we partition the data and run multiple copies of the model, which synchronize to exchange model weight updates. We have two common ways of implementing data parallelism: With a **parameter server** and with **AllReduce**.
Using a **parameter server**, each worker has a copy of the full model and trains on a shard of the data set. After computing the forward and backward pass, each worker sends the updates gradients to the PS, which aggregates

these and computes the new weights. The updated model weights then are sent back to all workers.

With **AllReduce**, we have data parallelism without a parameter server. The idea is that the GPU workers update their weights collectively using reduce-scatter, where the GPUs send their weights to others, receive weights from others, sum these up and send the summations again to others, until each worker has the total summation of one parameter. Then we enter the all-gather phase, where all the sums are sent to the other workers, until each worker has the sum of every parameter.
Due to fast GPU-GPU connections, AllReduce is the most commonly used way of implementing data parallelism today.

5.1.2 Model parallelism

If the model is too large to fit on a single device, we need to partition the model and train a part of the model on each device. To improve hardware utilization, we can use a pipelining approach using micro-batches.

5.2 Synchronous or asynchronous ML trainign

Using **synchronous** training, all the workers operate in lockstep to update the model weights at each iteration, i.e. a worker has to wait for the other workers to finish the iteration before it can start the next iteration. This usually leads to better model quality, but with a lower throughput.
When using **asynchronous** training, each worker independently updates its weights, without waiting for the other workers. Here, we have a higher throughput, but achieve lower model quality.

6 Communication APIs

6.1 Remote procedure calls (RPC)

Using RPC, we can turn a procedure call into a remote procedure. For the programmer, it still looks like calling a local procedure. For RPC, we need the following key steps:

- **Calling a procedure**: The call to the procedure is replaced by a "stub", which presents the same interface but manages the remote call.
- **Bind**: We need to be able to find the remote procedure.
- **Marshalling & serialization**: Map the parameters to a format suitable for transmission (usually string of bits).

RPC has the following advantages:

- Provides a mechanism to implement distributed systems in a simple way.
- RPC followed the programming techniques of its time (procedural languages), which made its adoption by programmers easier.
- RPC allows modular and hierarchical design of large distributed systems.

However, it also has some downsides:

- RPC is not a standard, but more an idea which has been implemented in various (incompatible) ways.
- RPC was designed with only one type of interaction in mind: client/server.
- Remote procedure calls are always slower than local procedure calls and remove procedures might not even return (e.g. when the other server is down).
- RPC also results in tight coupling of the involved systems.

To improve RPC, we can **attach an unique identifier to each call**. The client remembers the call it made, and guarantees **at least once** semantics (i.e. the call is repeated until it succeeds. The server also remembers the calls and guarantees **at most once** semantics (i.e. repeated calls are not executed twice). If we combine both, we have **exactly once** semantics.

6.2 Representational state transfer (REST)

RPC is highly efficient, but it becomes problematic at large scales and in the presence of failures. As HTTP became more prominent, the new concept of **REST** started to appear, with the following key features:

- REST is a general concept, with its most common implementation being HTTP.
- REST APIs describe **resources** instead of procedures, which can be anything.
- Resources have an unique identifier **URI**, which is similar to an URL.
- Data is exchanged using character formats, using standards like JSON.
- Interaction is **stateless**.

6.3 Design patterns to manage communication

6.3.1 Materialized view pattern

Most services just retrieve data and provide information to the caller, so instead of relying on a service, the caller caches a copy of the data it needs.

6.3.2 Service aggregator pattern

Instead of using independent services which can create a "complex web of calls" across other services, we use a service called the **aggregator** which enforces the sequence of calls.

6.4 Message queuing systems

In its most basic functionality, a **message queuing system** (MQS) provides queues for clients to PUT and GET messages, stores the massages persistently and makes the insertion & deletion of messages transactional. Persistent queues can be implemented as a part of a database, as a database system provides exactly the needed functionality.

6.5 Push subscribe

With standard client/server architectures and queuing systems, the client and server usually are aware of each other and they have a point-to-point interaction between them. In many situations, it's more useful to use a system where the interaction is based on announcing events:

- A service **publishes** messages/events of a given type
- Clients **subscribe** to different types of messages / events

7 Storage systems

7.1 Storage in unreliable systems

With the large scale of cloud clusters and the sheer number of storage devices involved, a cloud cluster sees many problems and failures over time.
However, the data must be available, and therefore storing the data in a single device would not work, as it would make the cloud very unreliable.

7.2 Replication

We have two basic parameters to select when designing a replication strategy: when the updates are propagated (**synchronous** vs **asynchronous**) and where the updates can take place (on the **primary copy** or **update everywhere**).

When we use **synchronous replication**, any change to the data is immediately propagated to all existing copies of the data. This approach is usually expensive in terms of latency, but experience over the years shows that it's better than the alternatives (hides the fact that the data is replicated from applications) and therefore is used by almost all data storage systems.
Asynchronous replication first executes the updates on the local copy, and only after that the updates are propagated to all other copies. During that time, the data is inconsistent (also called **eventual consistency** in

the cloud). It's mainly used in systems that can tolerate the lack of consistency, but it helps with performance and scalability.

Using **update everywhere**, changes can be initiated at any of the copies. On the other hand, using the **primary copy** approach, only one copy can be updated (the master), all others (secondary copies) are updated reflecting the changes to the master.

7.3 CAP theorem

In a distributed system, we can only have two of these three properties: consistency, availability and partition tolerance.

8 Reliability

Reliability is the ability of a system to remain operational over time. It is influenced by many factors:

- **Errors**: The system is operational, but produces wrong results.
- **Faults**: The system is operational, but some parts of it might not be.
- **Failures**: The system is no longer operational.
- **Performance issues**: The system is working but it's too slow to be useful.

8.1 Mean time to failure (MTTF)

- During an interval of time T, we observe F failures.
- Frequency of failures: F/T
- MTTF $= T/F$
- The failure rate is $1/MTTF$

The MTTF is an upper bound, real failures are likely to happen more often.
With $MTTF_{one}$ being the MTTF in one system, we have $MTTF_N = MTTF_{one}/N$ when using N systems, i.e. the probability that one node is down increases with N.

8.2 Mean time between failures (MTBF)

If the system can be repaired, the MTBF is the relevant measure. We have $MTBF = MTTF + MTTR$, where MTTR is the **mean time to repair**, i.e. the time between a failure occuring and the system being operational again.

The **availability** is equal to $MTTF/(MTTF + MTTR) =$ Time system is working / total time.

8.3 System configurations

Using **n-way** replication, we replicate the system on n nodes, where the system will be available if any of the replicas is vailable. To calculate the system availability, we take the product of $1 - a_i$, where a_i is the availability of system i, and subtract that number from 1. Usually, the more replicas, the higher the system availability.

We have to differ this from e.g. **triple module redundancy**, where we have 3 components, and the system is available as long as 2 components are available.

8.4 RAID levels

In RAID architectures, we use **striping** and **parity** to deal with failures in disk arrays. We can either have these **bit-interleaved**, i.e. each block is one bit, such that we e.g. can store a byte on 8 disks, where parity then is on a per byte basis. Or we can have **block-interleaved**, where each block contains several bytes (up to kbytes), parity then is on a per block basis.

8.4.1 RAID 0

RAID 0 strips data across disks but without adding any redundancy. While the bandwidth is greatly improved and the implementation in software is easy, a failure in one of the drives makes the entire RAID unavailable.

8.4.2 RAID 1

Bit interleaved, where we have fault-tolerance by mirroring without using parity. 50% of the disk capacity is used for redundancy purposes, I/O bandwidth is only half of RAILS 0. Recovery from failures is trivial, and can handle multiple disk failures (as long as they are not on a mirrored pair).

8.4.3 RAID 2

Bit interleaved, fault tolerance is provided by mirroring based on hamming codes (which implement parity for overlapping segments of data). Recovery is more complex, it can handle multiple disk failures (depending on the failures).

8.4.4 RAID 3

Bit interleaved, with a disk devoted to store the bit-wise parity of the other disks. I/O bandwidth is better than with level 1 and 2, recovery is relatively simple, tolerates one disk failure.

8.4.5 RAID 4

Block interleaved, with a disk devoted to store the block-wise parity of the other disks. Tolerates one disk failure, similar to RAID 4, but with coarse grain stripping (e.g. useful for standard databases). Write operations are sequential (as they all need to update the parity disk). Tolerates one disk failure.

8.4.6 RAID 5

Block interleaved, with block-wise parity distributed uniformly across all disks. Write operations can be done in parallel. Overall good performance, tolerates one disk failure.

8.4.7 RAID 10

First uses a RAID 0 controller to strip the data, each striping unit is then mirrored by a RAID 1 controller. Same fault tolerance as RAID 1, requires at least 4 disks, I/O bandwidth can be slightly better than RAID 1.

8.5 RAID 53

Uses a RAID 0 controller to strip the data and then gives each striping unit to a level 3 controller.

8.5.1 RAID 0 + 1

Uses a RAID 1 controller for mirroring the data and then a RAID 0 controller for striping the data.

8.6 Copyset replication

A **copyset** is a set of all nodes containing all copies of a data chunk. It's a single unit of failure, i.e. if we lose a copyset, we lose the data. Randomly placing the copysets performs bad, as we eventually create the max number of copysets, i.e. the probability of a failure is larger. Using MinCopysets, we minimize the number of copysets. In such a way, we minimize the probability of data loss, but if we lose data, we lose more data.

9 Server CPU design

9.1 Speedup recap

- **Amdahl's law**: fraction f accelerated by factor S
- **Speedup**: $\frac{CPUTime_{old}}{CPUTime_{new}} = \frac{1}{(1-f)+\frac{f}{s}}$
- Power usage: $capacitance \cdot voltage^2 \cdot frequency$
- Execution time: $\frac{instructionCount \cdot CPI}{frequency}$

9.2 Brawny vs wimpy cores

Brawny cores are high-end cores, which provide more performance, but also consume a lot more power. **Wimpy cores** are low-end cores, which provide less performance, but also use much less power.
An argument for brawny CPUs is that high per-core performance is important, as the performance is limited by the serial part of a program (Amdahl's law). For wimpy CPUs, the argument is that while they do provide less performance (e.g. $2x$ less), they use way less power (e.g. $5x$ less).

9.3 Number of sockets

When building a server, we have to decide whether to use a server with more or less sockets. Using more sockets, we have more memory capacity and amortize the cost of other components. Using less sockets, we have a closer integration of the cores and a more uniform memory access.

If we have only *light communication* between the servers, using many nodes has a relatively small performance overhead. However, if we have *medium to high communication*, we have a significant performance overhead for using many nodes.

The primary issue with using big servers with many sockets is that data center applications are often too large for even the largest servers. If we need to go outside of a big server, the penalty is pretty much the same as having many small servers. Also, clusters of small servers are often more cost effective, that's why today, a 2-socket server is considered a sweet spot.

9.4 Communication overhead

The execution time with communication overhead can be computed as:

$$\text{Execution time} = compute_{local}\ ms + f \cdot (comm_{local} \cdot \frac{1}{n} + comm_{remote} \cdot (1 - \frac{1}{n}))$$

where n is the number of nodes, $compute_{local}$ the local compute time, $comm_{local}$ the local communication time (same node) and $comm_{remote}$ the remote communication time to other nodes.

9.5 Cache allocation technology (CAT)

Intel **cache allocation technology** enables partitioning the ways of a highly associative last-level caching into several subsets with smaller associativity. Cores assigned to one subset can only allocate cache lines in their subset on refills, but are allowed to hit (read) in any part of the last level cache.

9.6 Power management

Cores can have multiple **power states**. The so called **p-states** describe different levels of performance / power consumption while executing codes. The **c-states** describe different power consumption/wake-up times while idling.

The p-states are for active power management, to balance the trade-off of power usage and performance.

9.6.1 Dynamic Voltage/Frequency Scaling (DVFS)

DVFS is used in p-states for active power management. So select the state, we set the frequency to the lowest needed to achieve the desired execution time:

$$\text{Execution time} = instruction\_count \cdot CPI \cdot clock\_time$$

10 Storage & Memory

10.1 SRAM & DRAM

Static Random Access Memory (SRAM) are preferable for registers and L1-L3 caches. They have fast access, no need to refresh, simpler manufacturing, but also a lower density and higher cost. **Dynamic Random Access Memory (DRAM)** are preferable for stand-alone memory chips. They have a much higher capacity than SRAM, a higher density and lower cost, but are also slower.

DRAM memory is organized into channels, ranks, banks and chips. For a read access, we have the following sequence:

- Decode the row address
- Selected bits drive the bit-lines, i.e. we read the entire row to the memory cache
- Amplify the row data
- Decode the column address and select the subset of the row
- Send to output

The DRAM memory-access protocol has 5 basic commands:

- ACTIVATE (open a row)
- READ (read a column)
- WRITE
- PRECHARGE (close row)
- REFRESH

When we access a **open row**, we don't need the ACTIVATE command, as the row is already in the buffer. We can directly READ/WRITE to the buffer. An access to a **closed row** first requires a PRECHARGE.

The **DRAM controller** has the following functions:

- Translate memory requests into DRAM command sequences
- Buffer and schedule requests to improve performance
- Ensure correct operation of DRAM (refresh)
- Manage power consumption and thermals

10.2 Flash storage

In flash storage, a **page** is the minimum unit of read/write operations. A **block**, which contains multiple pages, is the minimum unit of erase operations.
In a **write** operation to a page, only transitions from $1 \rightarrow 0$ transitions are allowed. Writing within a block must be ordered by page. To write to a page, it must be **erased** before it can be written. In flash storage, update-in-place is not possible. Reading takes significantly less time than writing & erasing.

As raw flash storage is not really useful, we have a **flash translation layer (FTL)** between the software and the flash storage. It maintains a map of physical page addresses to blocks/pages, and a block info table with some infos & statistics about the blocks.

11 Cloud security

11.1 Information security triad

We want the following:

- **Confidentiality**: Do not disclose information to unauthorized users
- **Integrity**: Do not allow unauthorized users to modify information
- **Availability**: The service or system should be available for use

11.2 Secure boot & trusted platform module

Secure boot is a mechanism that checks the system integrity before running any software, and certify the integrity to external users. It's implemented on a specialized chip, which stores encryption keys specific to the host system for hardware authentication.

11.3 Memory safety

For memory, we want **memory safety**, in form of **spatial memory safety** (ensuring all memory accesses are within bounds of the type being accessed) and **temporal memory safety** (ensure that pointers still point to valid memory at the time of dereference).
One way of ensuring memory safety is to write the program in a memory-safe language, e.g. Rust. However, this often requires re-writing all existing code, which is not always possible, ans as such, multiple hardware support for memory safety has been implemented.

11.3.1 ARM memory tagging extension (MTE)

ARM memory tagging extension is a lock-key mechanism to detect spatial & temporal memory safety violations.
A **lock** operation tags ("colors") memory with 4 bits of metadata for each 16 bytes of physical memory. Pointers (i.e. virtual addresses) containing the key ("color") in the top byte.
Due to a limited number of tag bits, it cannot guarantee safety, however it can catch most violations. It's useful for finding & debugging memory safety violations, but malicious programs can just brute-force the colors, since we only have 16 possible colors.

11.3.2 Intel memory protection keys (MPK)

Intel memory protaction keys enable a process to partition its memory into up to 16 regions and selectively disable or enable read/write access to each region. Again, it's not scalable, as we only have 16 regions.

11.4 Capabilities & Access control list

A **capabilities** is:

- A way of controlling access to resources in a system
- A token/key that gives whoever owns it the permission to access an object in a computer system.
- It consists of an unique object identifier and associated access rights.
- Finally, it should be transferable and protected from forgery.

The differences to access control lists are:

Capabilities:

- Track list of capabilities for each user
- Users can transfer capabilities
- Hard to revoke access
- Analogy: Normal keys to doors

Access control lists:

- Track list of users for each object
- The system is involved in access updates
- Easy to revoke access to an object
- Analogy: Card access sytem with central controller

11.5 Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions are extensions to Intel x86 processors that support:

- **Enclaves**: running code and memory isolated from the rest of the system
- **Attestation**: prove to local/remove system what code is running in the enclave
- **Minimum trusted computing base**: Only the processor is trusted, nothing else (DRAM & peripherals are untrusted, i.e. all writes to memory are encrypted)

The **enclaves** are isolated memory regions of code and date, which are written encrypted to main memory. The processor prevents access to cached enclave data outside of the enclave. Enclaves can only run unprivileged code (no system calls).