

263-3855-00: Cloud Computing Architecture

Section 13: Cloud Security

Swiss Federal Institute of Technology Zurich (ETH Zürich)

Last Edit Date: 05/31/2025

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Ana Klimovic and people from ETH Zurich Systems Group*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

I. Cloud security basic

1. Isolation

The ability to keep multiple software separated so that each instance only sees (confidentiality) and can affect (integrity) itself. Examples: browser implements site isolation, operating system provides process isolation, hypervisor provides VM isolation.

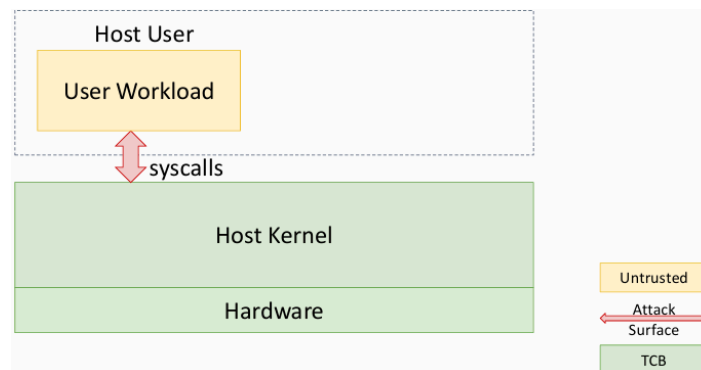
2. Trusted computing base (TCB)

All components (software and hardware) that are critical to the system's security: CPU and hypervisor are in the TCB of a cloud VM; the rest depends on your perspective, e.g., the guest application running in a cloud VM also trusts the guest OS inside the VM

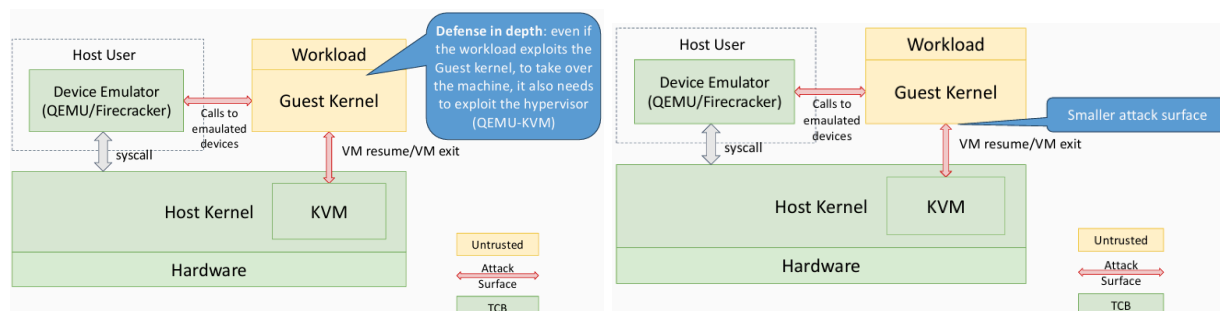
3. Attack surface

Set of interfaces through which an untrusted user cloud attempts to break security. The attack surface of a cloud VM includes: CPU instruction set (in VM guest mode), hypercall API (calls VM can make to hypervisor), virtual I/O devices, guest-facing network services

4. TCB and attack surface example - containers



5. TCB and attack surface example - Firecracker MicroVM



6. Threat model

- A set of assumptions about

- What components (hardware & software) you can trust or fully control, for example, CPU and hypervisor are trusted, network is untrusted
- What the “attacker” can and cannot do
 - Can run arbitrary user code
 - Cannot break encryption, cannot bypass CPU memory protection
- Define scope when evaluating security claims (in an academic context)

Typical threat model for cloud computing

- Trust relationships
 - Everyone trusts the hardware
 - Users do not trust each other
 - Users trust the cloud provider (and its software)
 - Cloud provider does not trust users
- Who has access to what?
 - Users do not have physical access to hardware
 - Cloud provider has physical access

II. Side channels

1. Security challenges with shared hardware

- Shared hardware opens the door to numerous side channels
 - Memory access patterns
 - State of processor caches
 - State of branch predictor
- A side channel is a source of unintended communication between isolated components
 - Can lead to leaking private information

2. Side channels & covert channels

- Side channel: unintended communication between isolated components
 - A victim can unintentionally leak private information to an attacker
- Covert channel: hidden channel used to get information out
 - Two parties want to communicate without others knowing the communication is happening
- Covert channel example: signal behavior across VMs
 - Assume 2 VMs share the same underlying hardware
 - To send a bit $b \in \{0, 1\}$ malware does:
 - $b = 1$: at 1:00 am do CPU intensive calculation

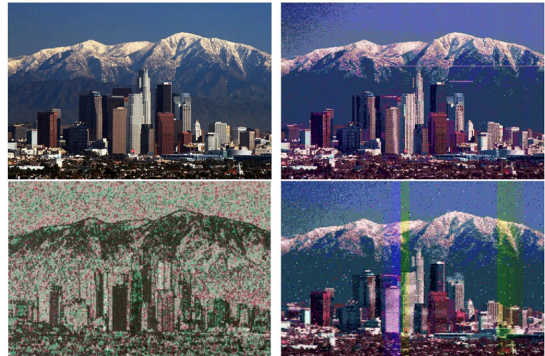
- $b = 0$: at 1:00 am do nothing
- At 1:00 am listener does CPU intensive calculation and measures completion time
 - $b = 1 \Rightarrow$ completion time $>$ threshold

How do side / covert channels work?

- Extract info from a system (process of VM) by observing usage or noise patterns on shared resources
 - Info leakage may be slow but key information is often small, for example, passwords or type of program
- Countless practical attacks demonstrated
 - Based on timing patterns, cache patterns, power analysis, electromagnetic emissions, acoustic emissions, ...
 - Some of them require just a co-located VM

Simple example: timing attack

- Assume a program receives a password string as input and validates whether it matches the secret
- This program analyses the provided string one character at a time, comparing each character to the corresponding character of the secret
- Returns an error as soon as it encounters a mismatch
- The program “leaks” information about the secret: it has different response time for an input that starts with one or more of the same characters as the secret vs. for an input that does not
- Through trial and error, an attacker can determine the value of the secret one character at a time by measuring the time to response to certain inputs



JPEG image recovery attack example

An untrusted OS can recover a JPEG image from a user app running in an SGX enclave via page table and cache side channels

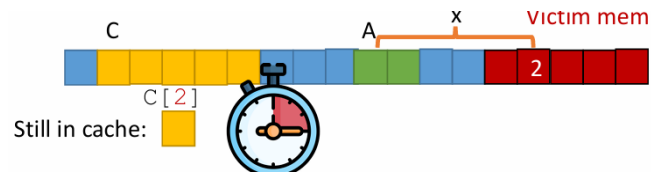
Spectre side channel attacks

- Spectre: a large class of side channel attacks
 - Enabled by speculation mechanisms in CPU hardware
 - Present in vast majority of modern processors
 - Hard to fix
 - Ultimate fix: change processor
 - Variety of software fixes: some with high cost (degrade performance)

- Fundamental issue with instruction set architecture (ISA) definition
 - ISA preserves functional behavior, not performance behavior
 - Side channels use performance behavior to leak info

Spectre: general structure of attacks

- Step 1: train some microarchitectural state, e.g.
 - Clear 256 blocks in cache at address C (as there are 256 ASCII characters)
 - Prime branch predictor for a branch to “taken”
- Step 2: Save secret in the microarchitecture, e.g., cache state:
 - if ($x < \text{size_A}$): $y = C[A[x]]$

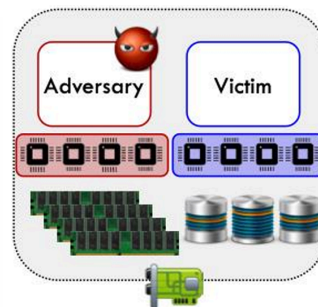


- Step 3: Extract secret from the microarchitecture, e.g., cache state:
 - Time accesses to 256 cache blocks at C; the block that hits (and is faster) corresponds to the value of byte at address A[x]

Other types of attacks with side channels

- **Idea:** apply cache, memory, CPU, I/O interference to detect which type of app is collocated on the same VM. Then can execute target attacks.
- Information about app leaked, but not data

Application class	Detection accuracy (%)
In-memory caching (memcached)	80%
Persistent databases (Cassandra, MongoDB)	89%
Hadoop jobs	92%
Spark jobs	86%
Webservers	91%
Aggregate	89%



AWS EC2 approach to side channels

- Physical CPU cores are never simultaneously shared between different customer VMs with simultaneous multi-threading
 - This also means that customer VMs do not share CPU registers or L1/L2 caches
 - Customers may share last level cache, but these attacks using last level cache are less practical (signal is more noisy)
 - AWS EC2 may share physical CPU cores sequentially, but context switching between different VMs -> need to clear state between context switches

III. Confidential computing

- Goal: guarantee confidentiality and integrity of a user's computation running on a remote (cloud) system, including:
 - The program
 - Its inputs and outputs
 - Intermediate state, control flow, etc.
- Even if do not trust the cloud provider

Threat model for confidential cloud computing

- Trust relationships
 - Everyone trusts the hardware
 - Users do not trust each other
 - Users **do not** trust the cloud provider (and its software)
 - Cloud provider does not trust users
- Who has access to what?
 - Users do not have physical access to hardware
 - Cloud provider has physical access

Why is this hard?

- Need to protect data and code in any state that it exists in the cloud
- At rest (i.e., in storage) - standard solution: encrypt data
- In transmission (i.e., over the network) - standard solution: encrypt data
- In use (i.e., during processing, in memory)
 - Compute "trusted execution environments" - use hardware support
 - Compute on encrypted data (homomorphic encryption) - not (yet) practical for most applications

Trusted execution environments (TEEs)

- Trust some (hopefully small) remote hardware and software
- Trusted hardware element (e.g., TPM chip)
 - Measure (~hashes) privileged software
 - Attests (~signs has) to a remote user
- Incorrect attestation -> compromised software
- Allows the user to detect if TCB is tampered

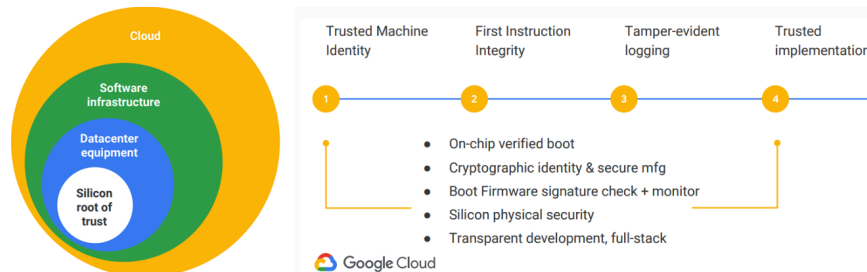
Secure boot with a trusted platform module

- Secure boot: check system integrity before running software

- Certify system to external user
- TPM is a specialized chip that stores encryption keys specific to the host system for hardware authentication
 - Goals: secure attestation & key management
 - Measure & examine state, send signed certification
- A trusted boot block is stored on the TPM
- It is used to measure the BIOS before running
 - Example: SHA-1 digest of the code
 - Does its measurement match expected signature
- Then measure the OS before running
 - Does its measurement match expected signature
- Then measure any software before running
 - Does its measurement match expected signature

Cloud example: Google's Titan chip

Secure low-power microcontroller designed with cloud security as first-class consideration



Key ingredients for confidential computing

- CPU root of trust
 - Cryptographic keys and secure boot
- Attestation infrastructure
- Privileged execution environment
 - Can be implemented with different forms, e.g.,
 - Extended microcode (e.g., Intel SGX enclaves)
 - Platform security processor (e.g., AMD SEV confidential VM)
 - Trusted firmware
 - New CPU protection model (e.g., Intel TXT confidential VM, ARM CCA confidential VM)
 - Trusted software (“monitor”) runs below a de-privileged hypervisor

Implementing security for cloud computing

Different strategies may rely on

- Cryptographic primitives
 - Encrypt data, authenticate remote users / systems
- Formal methods and verification
 - Prove your software / hardware has the expected behavior
- Hardware isolation mechanisms
 - Check identity, check permissions, and protect data / code at hardware level

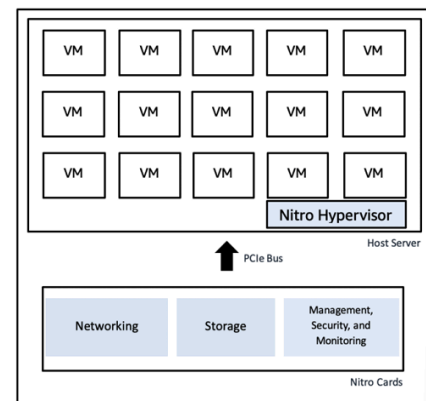
Hardware's role in security

- Implement security mechanisms
 - Fully or partially
- Reduce trusted code base
- Accelerate expensive software mechanisms
 - Example: AES (crypto) extensions
- Hardware alone is not sufficient, e.g., the best lock does not help if I give away the key

IV. Hardware support for security

1. Hardware-assisted hypervisor - AWS Nitro

- Nitro project is AWS's approach to redesigning hypervisor technology
 - Goal: improve security, performance, cost, and pace of innovation
 - Approach: offload the majority of hypervisor functionality to specialized hardware, independent of the main system CPU board
- Nitro is AWS's approach to redesigning hypervisor technology



NITRO CARDS



Hardware devices designed by AWS that provide overall system control and I/O virtualization independent of the main system board with its CPUs and memory.

NITRO SECURITY CHIP



Enables a secure boot process for the overall system based on a hardware root of trust, the ability to offer bare metal instances, as well as defense in depth that offers protection to the server from unauthorized modification of system firmware.

NITRO HYPERVISOR



A deliberately minimized and firmware-like hypervisor designed to provide strong resource isolation, and performance that is nearly indistinguishable from a bare metal server.

AWS Nitro cards

- Custom silicon implementing data encryption for networking and storage

- Use hardware offload engines with secure key storage integrated in the SoC
- Encryption keys are only present in plaintext in the protected volatile memory of the Nitro cards
 - Keys are inaccessible to AWS operators or customer code running on host CPUs
- A Nitro card can be subdivided into multiple virtual functions with SR-IOV

AWS Nitro security chip

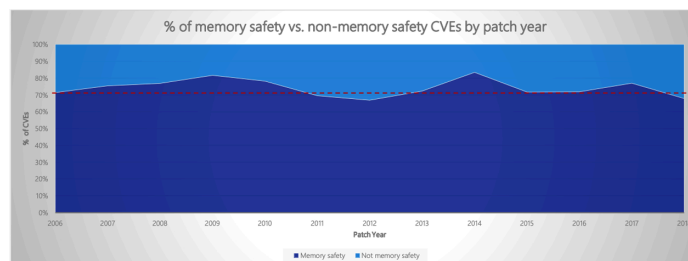
- Device integrated into the server's system main board
- During system boot / reset, controls physical reset pins
 - Ensures Nitro controller completes its secure boot and verifies integrity of the BIOS and system firmware before any processing on CPU can begin
- To support bare-metal EC2 VMs, Nitro security chip ensures that host CPU cannot update firmware in bare-metal mode

AWS Nitro hypervisor

- Nitro hypervisor code is a cryptographically signed firmware-like component stored on encrypted local storage
- By design, the hypervisor is not a general-purpose system: it has no network stack, no file system, no interactive shell
 - Relies instead on physically isolated Nitro cards for I/O
 - Eliminates entire classes of bugs, such as remote networking attacks
 - Even in the unlikely case of a bug, the lack of standard features like interactive shells and file systems makes it a difficult environment for attackers to exploit

V. Memory safety for secure code / data isolation

- Memory safety bugs are a major cause of security vulnerabilities
 - System code is often written in languages that are not memory-safe (e.g., C, C++)
 - Memory safety bugs can allow attackers to inject malicious code and escalate their privilege
 - According to Microsoft, 70% of security issues are due to memory safety violations



Memory safety

- Spatial memory safety

- Ensure all memory accesses are within bounds of the type being accessed
 - Example violation: access elements beyond the size of an allocated array
- Temporal memory safety
 - Ensure that pointers still point to valid memory at the time of dereference
 - Example violation: use after free
- Especially important when multiple software components are sharing memory on a machine

How to ensure memory safety?

Implement software in a memory-safe language (e.g., Rust)

But what if it is not practical to rewrite all existing code?

- Software fault isolation (SFI)
 - Insert bounds checks on each memory access (e.g., during compilation), e.g., WebAssembly implements SFI to sandbox user code
- Hardware support for memory safety
 - Memory management unit-based page protection (e.g., processes use different page tables, VMs add another layer of page transition)
 - Memory protection keys and Memory capabilities
 - Allow for lighter-weight context switching between memory-isolated components

Hardware memory capabilities

- What is a “capacity”?
 - A way of controlling access to resources in a system
 - A capability is a token / key that gives whoever owns the capability permission to access an entity / object in a computer system
 - A capability consists of a unique object identifier and associated access rights
- Capabilities are an alternative to access control lists (ACLs)
- Replace integer pointers with capabilities, i.e., “fat pointers” which contain not just an address but also bounds and permissions for memory regions
- To make capabilities a useful security mechanism, need 3 key properties
 - Provenance: capabilities can only be derived from other capabilities
 - Integrity: capabilities cannot be forged
 - Monotonicity: for derived capacities, permissions can only be reduced

Capabilities vs. Access control lists (ACLs)

- Capabilities:
 - Track list of capabilities for each user



- Users can transfer capabilities
- Hard to revoke access
- Access control lists:
 - Track list of users for each object
 - System involved in access updates
 - Easy to revoke access to one object

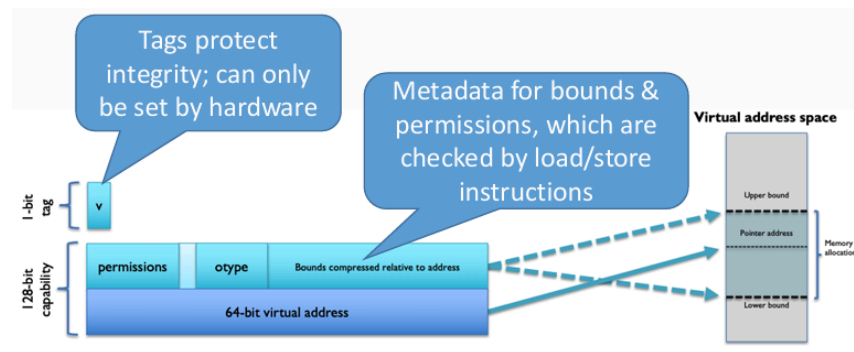


Analogy: If you want to give access to rooms in a building (“objects”), you can give people keys to the rooms (“capabilities”) or you can have a card access system where for each user you control which rooms their card can open (“access control list”)

Capabilities can be implemented in software or hardware, for controlling access to diverse resources (e.g., memory addresses, files on storage devices, etc.)

CHERI memory capabilities

CHERI project from the University of Cambridge and SRI international implements memory capabilities as ISA extensions for MIPS, RISC-V, ARM



Use cases for hardware memory capabilities

- Benefits / opportunities
 - Automatically catch and disallow spatial memory safety violations (e.g., buffer overflow)
 - Sandboxing and software compartmentalization; different parts of an application can securely isolate their code / data even if running in the same address space
- Challenges
 - Software compatibility; users may need to modify their code and / or recompile
 - Revoking capabilities is expensive. To revoke a capability, you need to invalidate all instances of it. Since we do not keep track of how capabilities are passed around, we need to sweep all memory pages to look for the capability