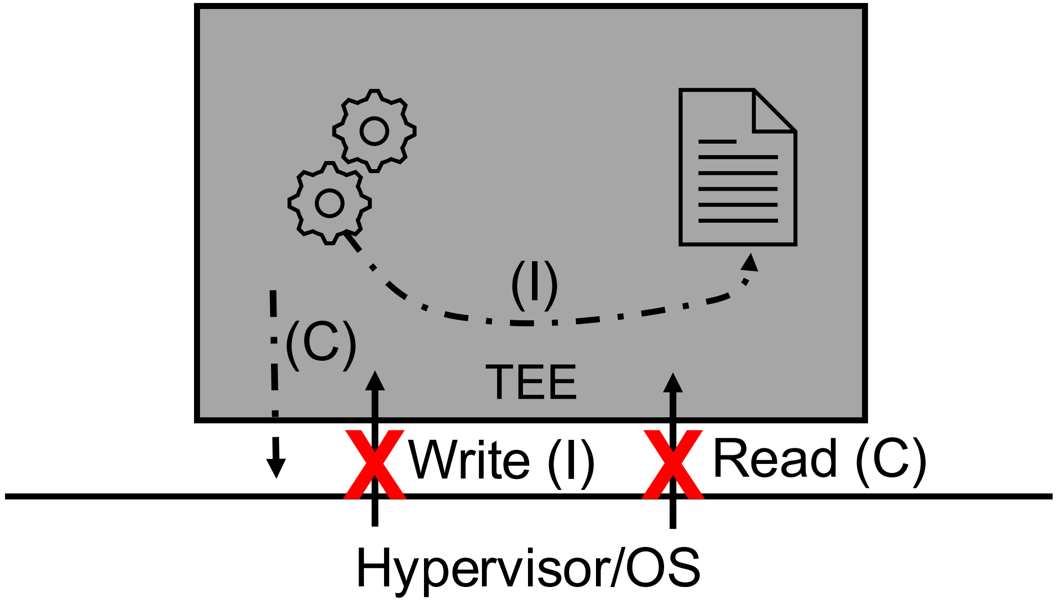
Why I Stopped Caring About the TCB

Adrien Ghosn
Azure Research
Microsoft

Marios Kogias
Imperial College London
IMPERIAL

TEEs and the Provided Guarantees

- Hardware-level support for:
  - Confidentiality (C)
  - Integrity (I)
  - against **external** attackers



How can we guarantee confidentiality and integrity against **internal** attackers?

TCB: A blast from the past

- Intel SGXv1 was among the first commercially available TEEs
- Used to isolate **parts** of an application
- Single address space

- The TCB by definition included **everything** that runs inside the TEE
- A **low TCB** was the only way to reason that SGX does not leak information
  - Auditing
  - Full formal verification

Towards cVMs and Usable CC

- **Confidential VMs** are becoming the de facto standard
- Different offerings by hardware vendors

intel — TDX | AMD — SEV-SNP | arm — CCA

- **Practicality**:
  - Run existing large codebases inside the TEE
  - Easier and more practical to deploy confidential applications
- More **isolation features**:
  - Traditional isolation mechanisms: protection rings and page tables
  - New isolation mechanisms: e.g., VMPLs in AMD

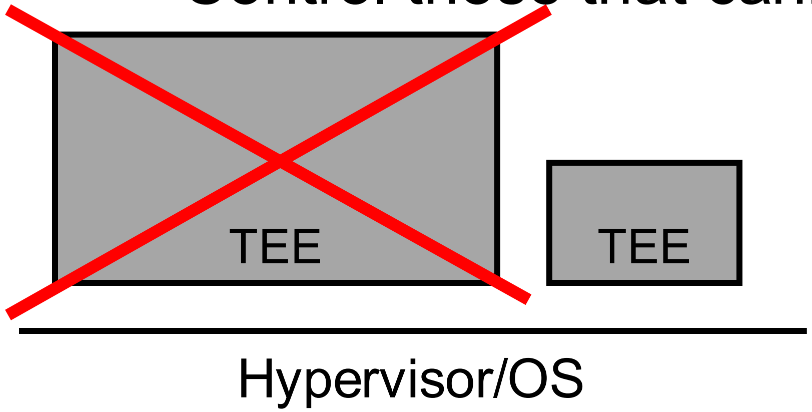
How should we think about the TCB in this new TEE world?

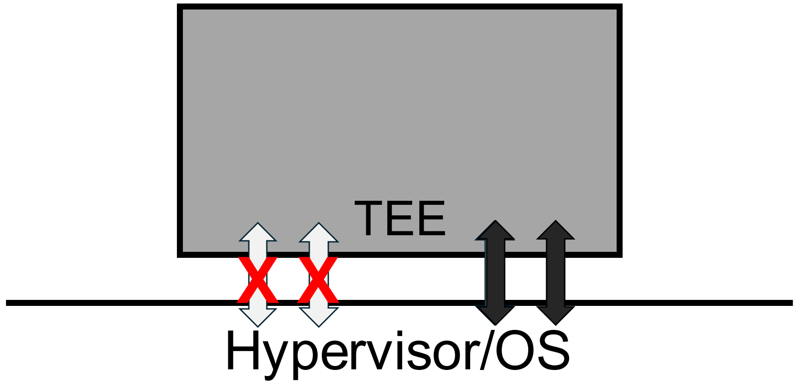
Positions

- TCB does **not** include all the code that runs inside a TEE
  - 👉 Stop caring about the TCB size or using it as a metric for security
- Internal **integrity** protection is not a TEE-specific requirement
  - 👉 Use existing compartmentalization mechanisms to tackle it
- Focus on **confidentiality**
  - 👉 Need a systematic way to ensure a TEE **does not leak** data

High-Level Idea

- Avoid reasoning about what the code inside the TEE can do
  - Small TCB requirement for auditing, formal verification... (old approach)
- Actively enforce what the TEE **should not do**
- Focus on the TEE **interfaces**
  - Enumerate all the communication channels
  - Eliminate those that are not necessary
  - Control those that cannot be avoided



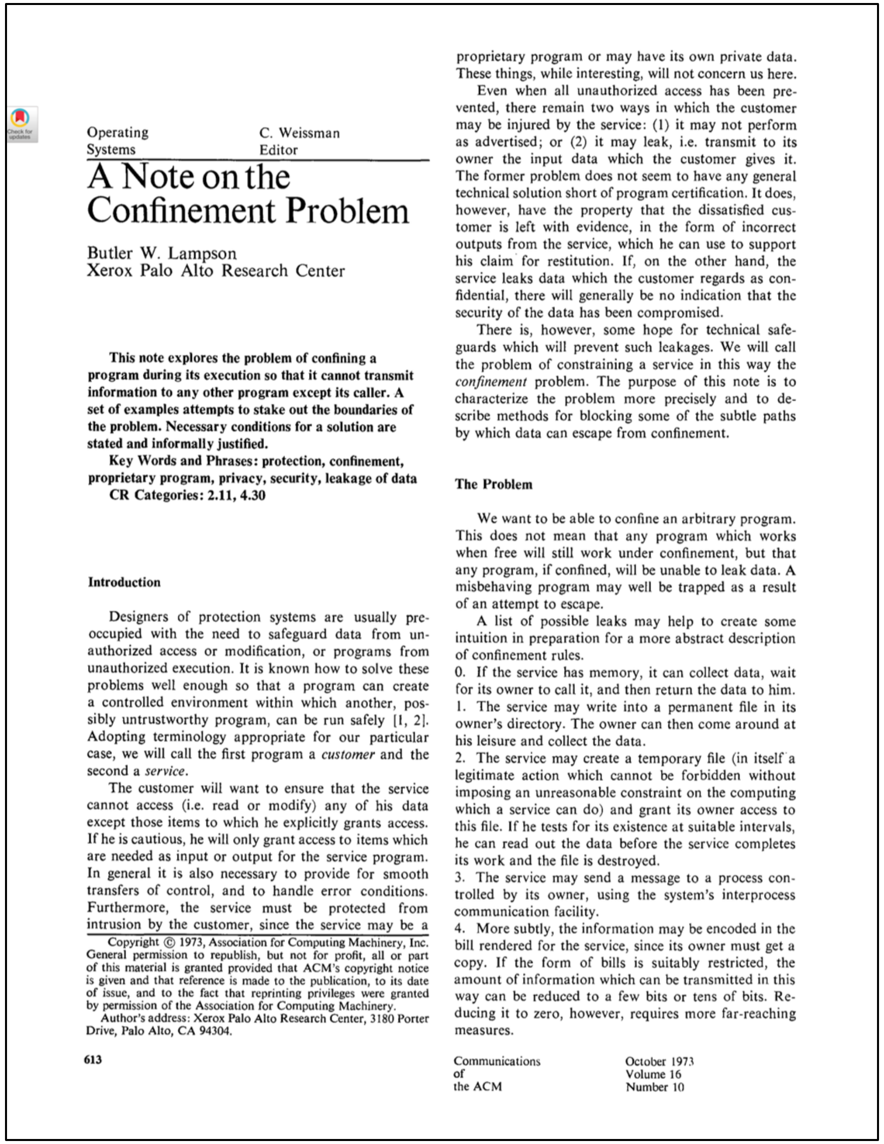


A Note on the Confinement Problem

- This is not a new idea/problem
- Published by Butler Lampson in 1973
- Introduced a set of “confinement rules”
  - *Masking:* A program to be confined must allow its caller to determine **all** its inputs into legitimate and covert channels.

Can we turn all confidentiality problems into confinement problems? **YES**

How can the confinement rules be applied in modern TEE architectures?



Operating Systems C. Weissman Editor

A Note on the Confinement Problem

Butler W. Lampson
Xerox Palo Alto Research Center

This note explores the problem of confining a program during its execution so that it cannot transmit information to any other program except its caller. A set of examples attempts to stake out the boundaries of the problem. Necessary conditions for a solution are stated and informally justified.

Key Words and Phrases: protection, confinement, proprietary program, privacy, security, leakage of data

CR Categories: 2.11, 4.30

Introduction

Designers of protection systems are usually preoccupied with the need to safeguard data from unauthorized access or modification, or programs from unauthorized execution. It is known how to solve these problems well enough so that a program can create a controlled environment within which another, possibly untrustworthy program, can be run safely [1, 2]. Adopting terminology appropriate for our particular case, we will call the first program a *customer* and the second a *service*.

The customer will want to ensure that the service cannot access (i.e. read or modify) any of his data except those items to which he explicitly grants access. If he is cautious, he will only grant access to items which are needed as input or output for the service program. In general it is also necessary to provide for smooth transfers of control, and to handle error conditions. Furthermore, the service must be protected from intrusion by the customer, since the service may be a proprietary program or may have its own private data. These things, while interesting, will not concern us here.

Even when all unauthorized access has been prevented, there remain two ways in which the customer may be injured by the service: (1) it may not perform as advertised; or (2) it may leak, i.e. transmit to its owner the input data which the customer gives it. The former problem does not seem to have any general technical solution short of program certification. It does, however, have the property that the dissatisfied customer is left with evidence, in the form of incorrect outputs from the service, which he can use to support his claim for restitution. If, on the other hand, the service leaks data which the customer regards as confidential, there will generally be no indication that the security of the data has been compromised.

There is, however, some hope for technical safeguards which will prevent such leakages. We will call the problem of constraining a service in this way the *confinement* problem. The purpose of this note is to characterize the problem more precisely and to describe methods for blocking some of the subtle paths by which data can escape from confinement.

The Problem

We want to be able to confine an arbitrary program. This does not mean that any program which works when free will still work under confinement, but that any program, if confined, will be unable to leak data. A misbehaving program may well be trapped as a result of an attempt to escape.

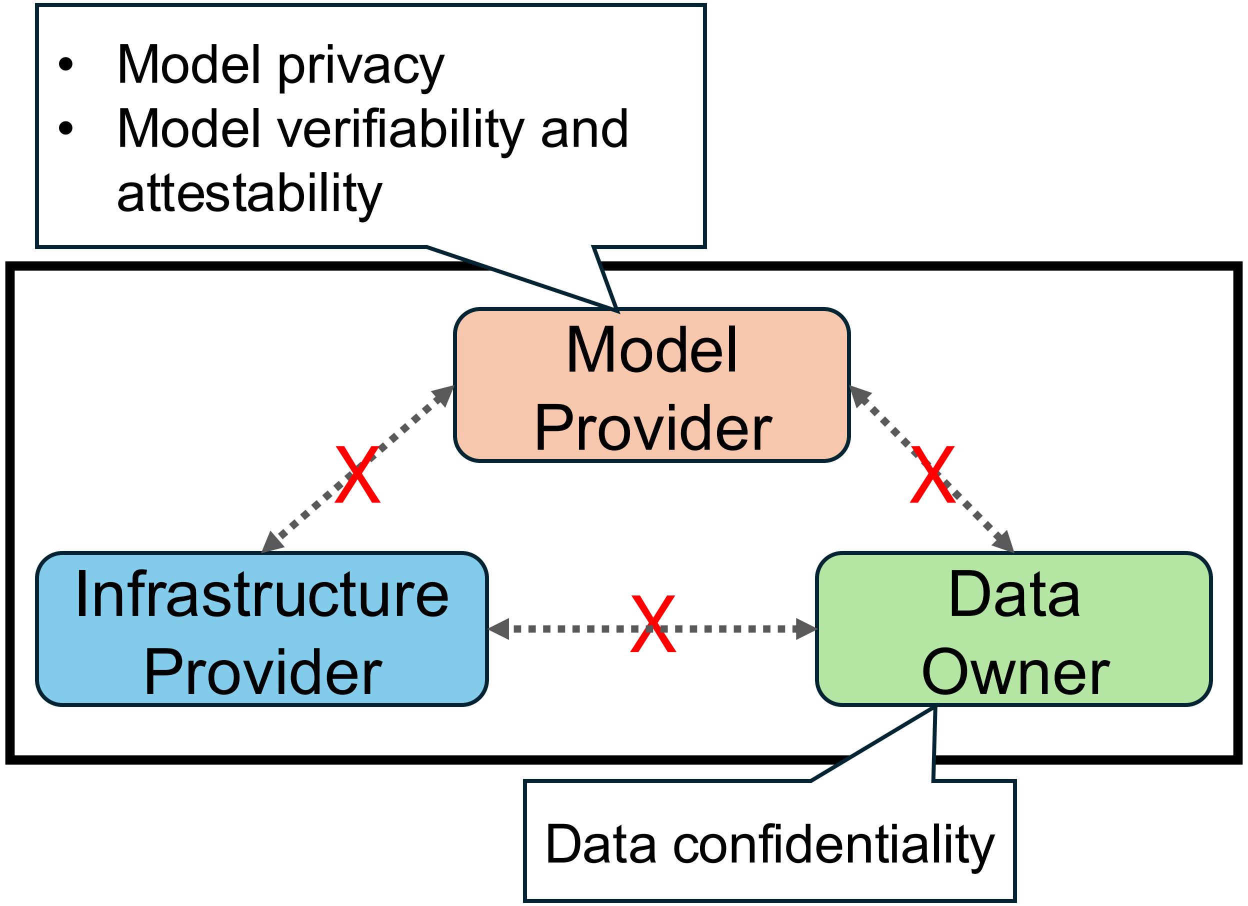
A list of possible leaks may help to create some intuition in preparation for a more abstract description of confinement rules.

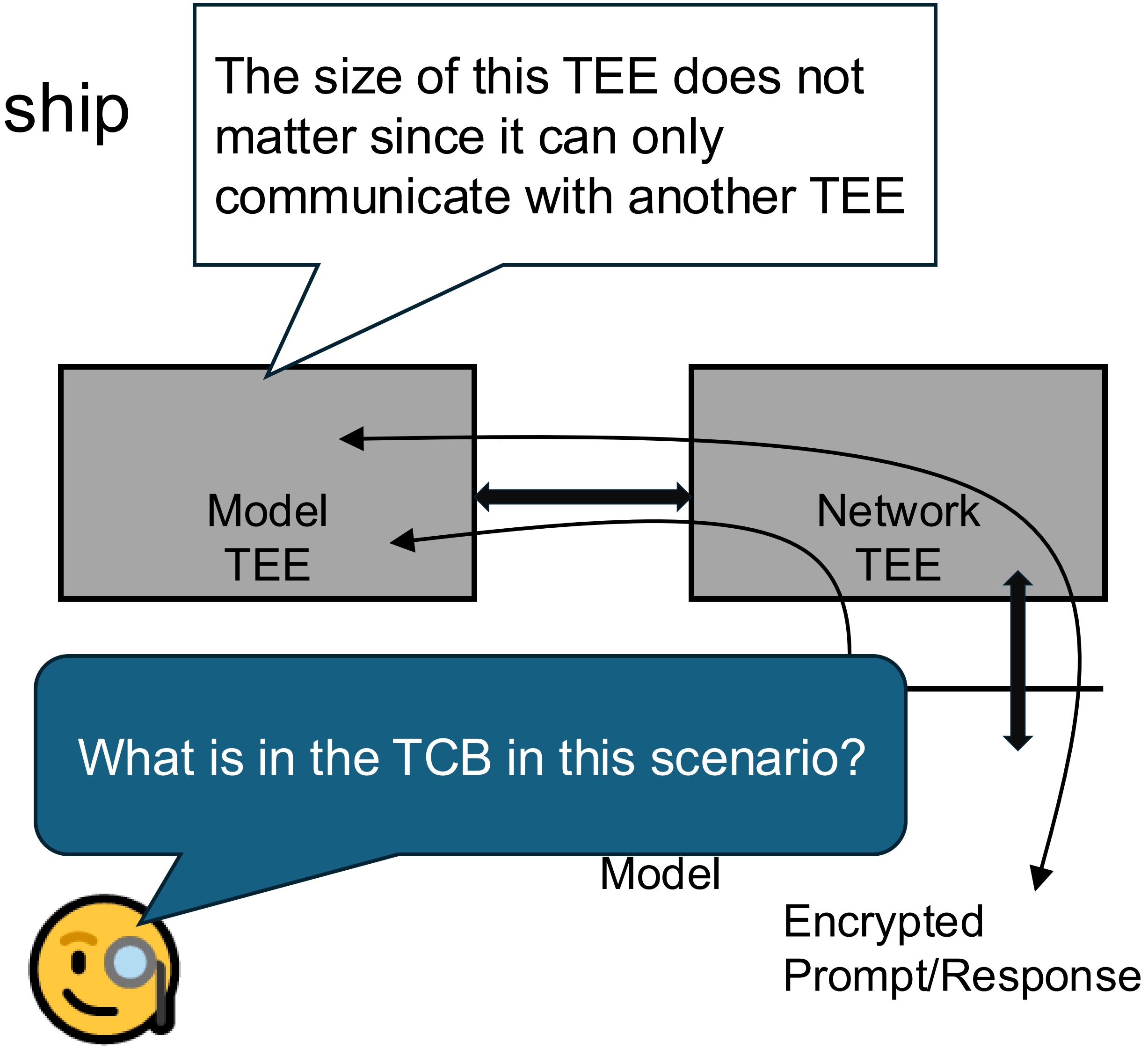
0. If the service has memory, it can collect data, wait for its owner to call it, and then return the data to him.
1. The service may write into a permanent file in its owner's directory. The owner can then come around at his leisure and collect the data.
2. The service may create a temporary file (in itself a legitimate action which cannot be forbidden without imposing an unreasonable constraint on the computing which a service can do) and grant its owner access to this file. If he tests for its existence at suitable intervals, he can read out the data before the service completes its work and the file is destroyed.
3. The service may send a message to a process controlled by its owner, using the system's interprocess communication facility.
4. More subtly, the information may be encoded in the bill rendered for the service, since its owner must get a copy. If the form of bills is suitably restricted, the amount of information which can be transmitted in this way can be reduced to a few bits or tens of bits. Reducing it to zero, however, requires more far-reaching measures.

Author's address: Xerox Palo Alto Research Center, 3180 Porter Drive, Palo Alto, CA 94304.

Example: Confidential ML

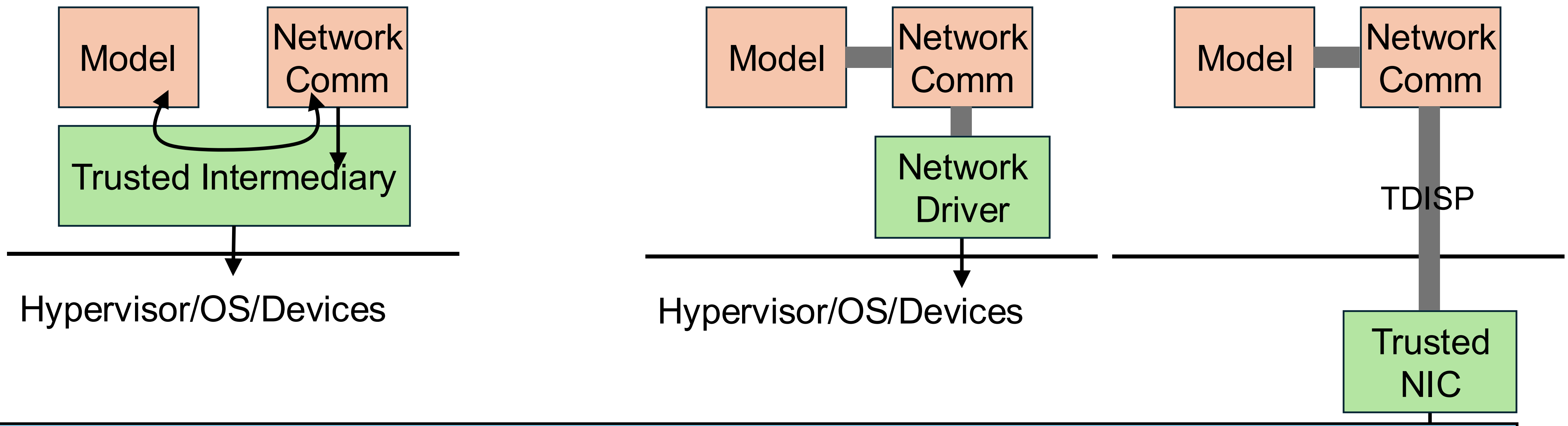
- Interesting tripartite trust relationship





Designs for Securing Interfaces

- Trusted Intermediary
- Trusted Channels



The architecture and the TCB changes based on the hardware capabilities

What is the minimum hardware support?

- A form of **physical isolation**
  - Page tables
  - VMPLs
  - Other piece of hardware (e.g. separate NIC)

- At least two **privilege levels**:
  - To dynamically check interactions as a trusted intermediary
  - To statically configure the trusted channels

Conclusion

- The TCB does **not include everything** that runs inside the TEE
- Confidentiality can be seen as a **confinement** problem
- Proposal: Focus and secure the TEE **interfaces**
  - Open challenge how to secure existing interfaces
  - Open challenge how to design good interfaces for non-trusted parties
- TEE **hardware** is becoming more **elaborate**
  - Let’s use it to build better and more usable CC applications

Thank you!

m.kogias@imperial.ac.uk
https://marioskogias.github.io