Towards Application Security on Untrusted Operating Systems

Dan R. K. Ports
MIT CSAIL & VMware

Tal Garfinkel
VMware

Motivation

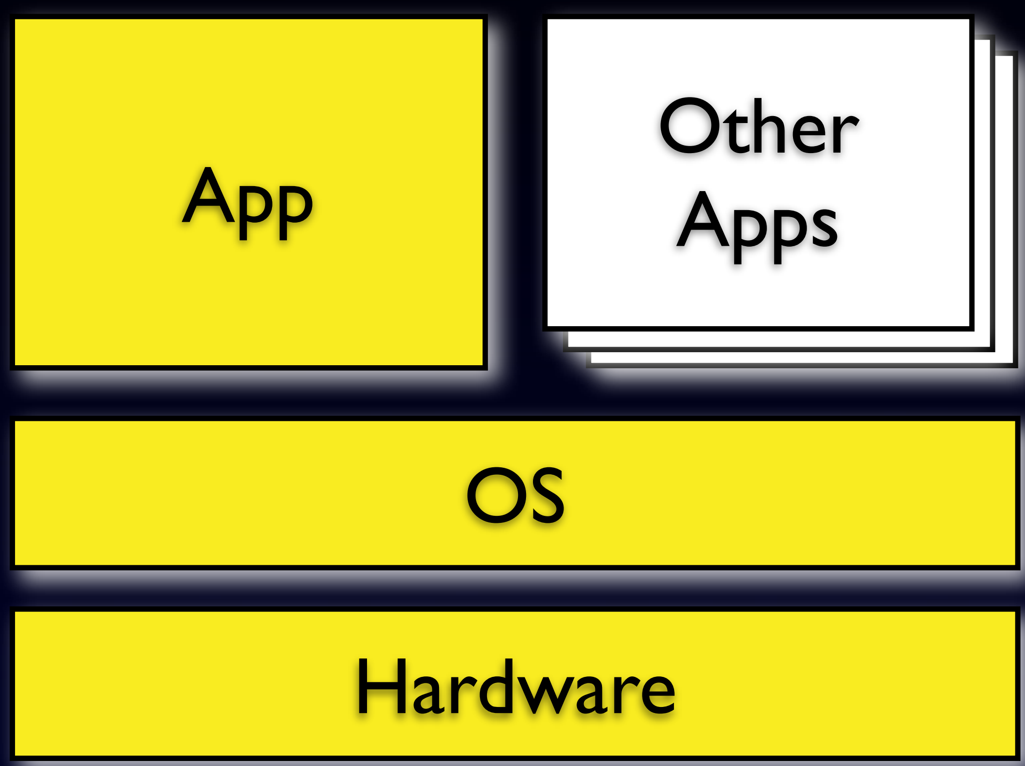
Many applications handle sensitive data

financial, medical, insurance, military...
credit cards, medical records, corporate IP...

...but run on commodity operating systems

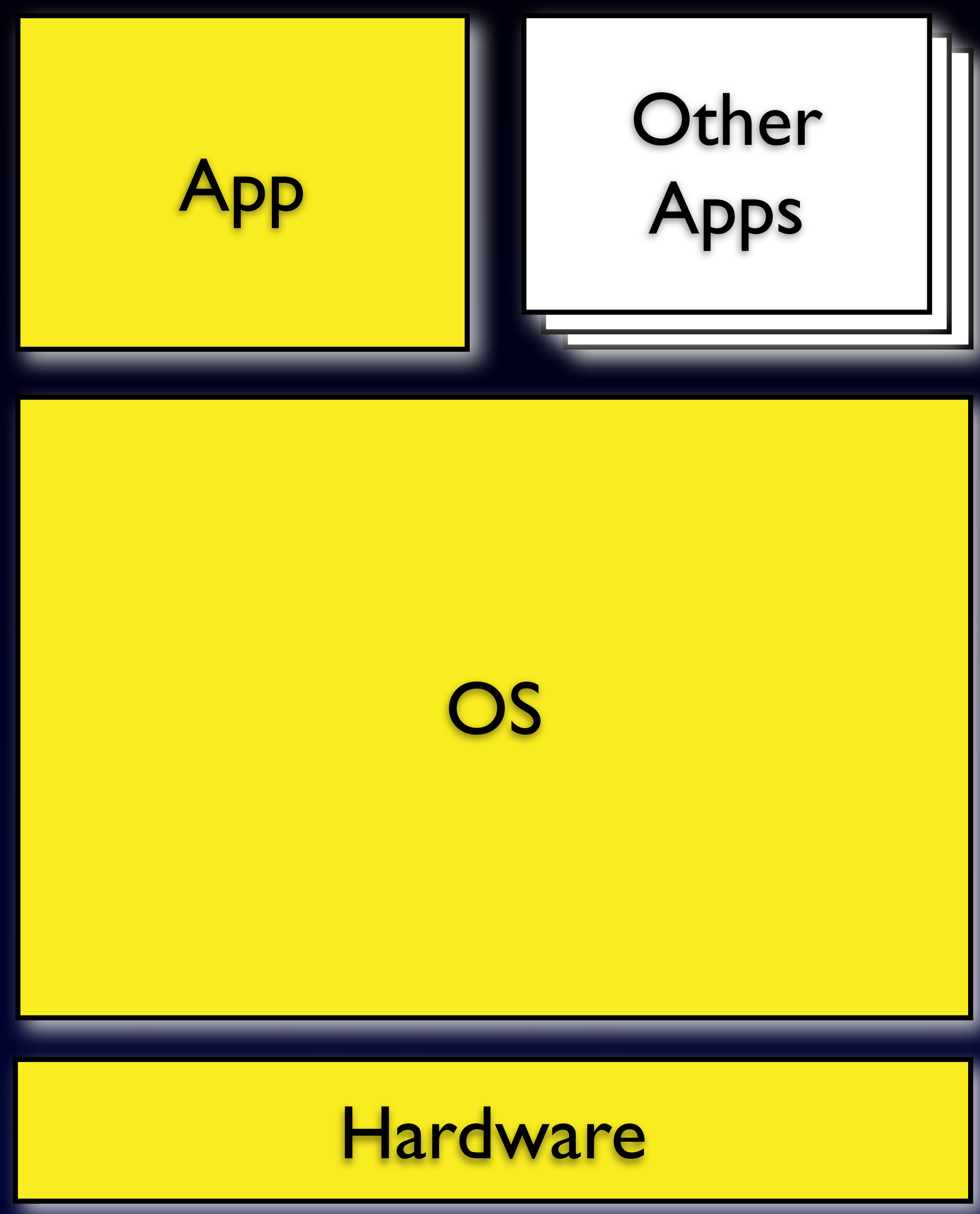
Complexity leads to poor assurance!

Large TCB Sizes



Theory:
few small trusted parts;
can be assumed correct

Large TCB Sizes



Theory:
few small trusted parts;
can be assumed correct

Reality:
OS has many trusted parts:
- kernel
- device drivers
- system daemons
- anything running as root

This is a problem.

This is a problem.

(and it’s not likely to solve itself)

Defense in Depth Approach

Defense in Depth Approach

Continue to use existing OS components, without fully trusting them

Add security layer to protect sensitive data even if OS is compromised

Defense in Depth Approach

Continue to use existing OS components, without fully trusting them

Add security layer to protect sensitive data even if OS is compromised

Desired security property:
apps always behave normally
even if the OS behaves maliciously

Defense in Depth Approach

Continue to use existing OS components, without fully trusting them

Add security layer to protect sensitive data even if OS is compromised

Desired security property:

apps always behave normally *(or fail-stop)* even if the OS behaves maliciously

Problem: OS solely responsible for CPU / memory resource management

➡ can access application memory & control application execution

Solution: isolated execution environment

➡ give app memory that OS can't access

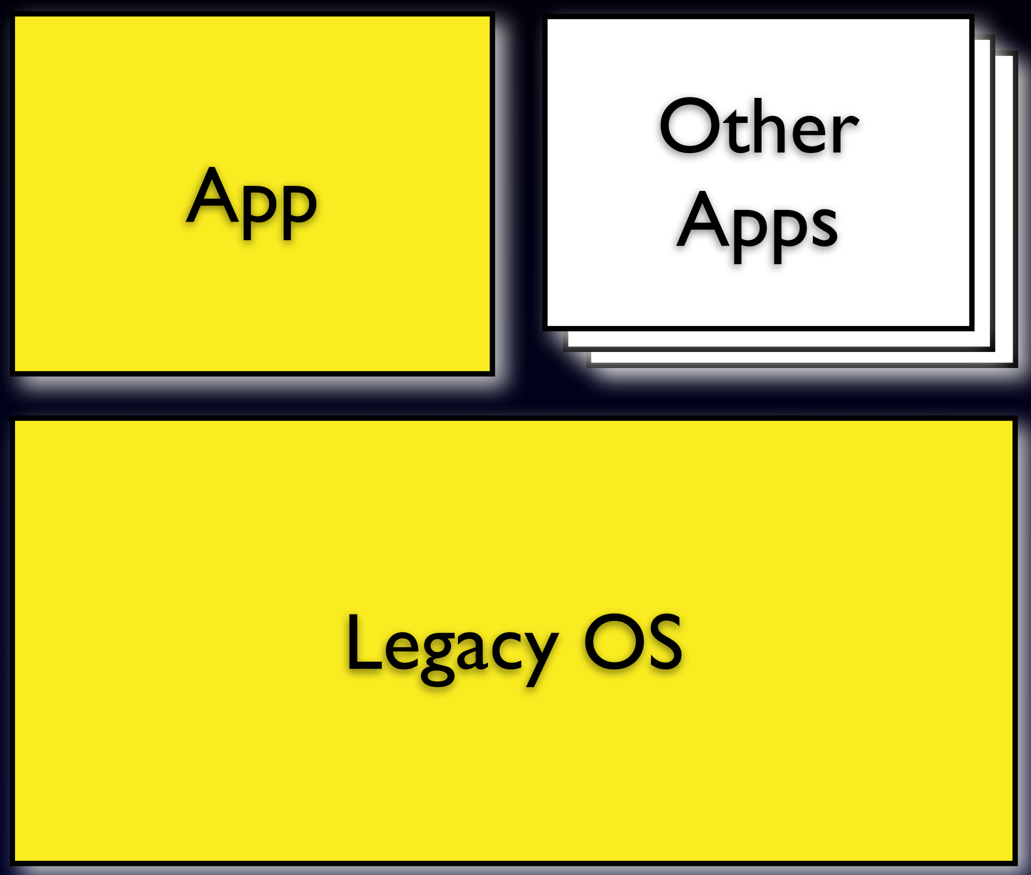
Is CPU & memory isolation enough to run apps securely on an untrusted OS?

Is CPU & memory isolation enough to run apps securely on an untrusted OS?

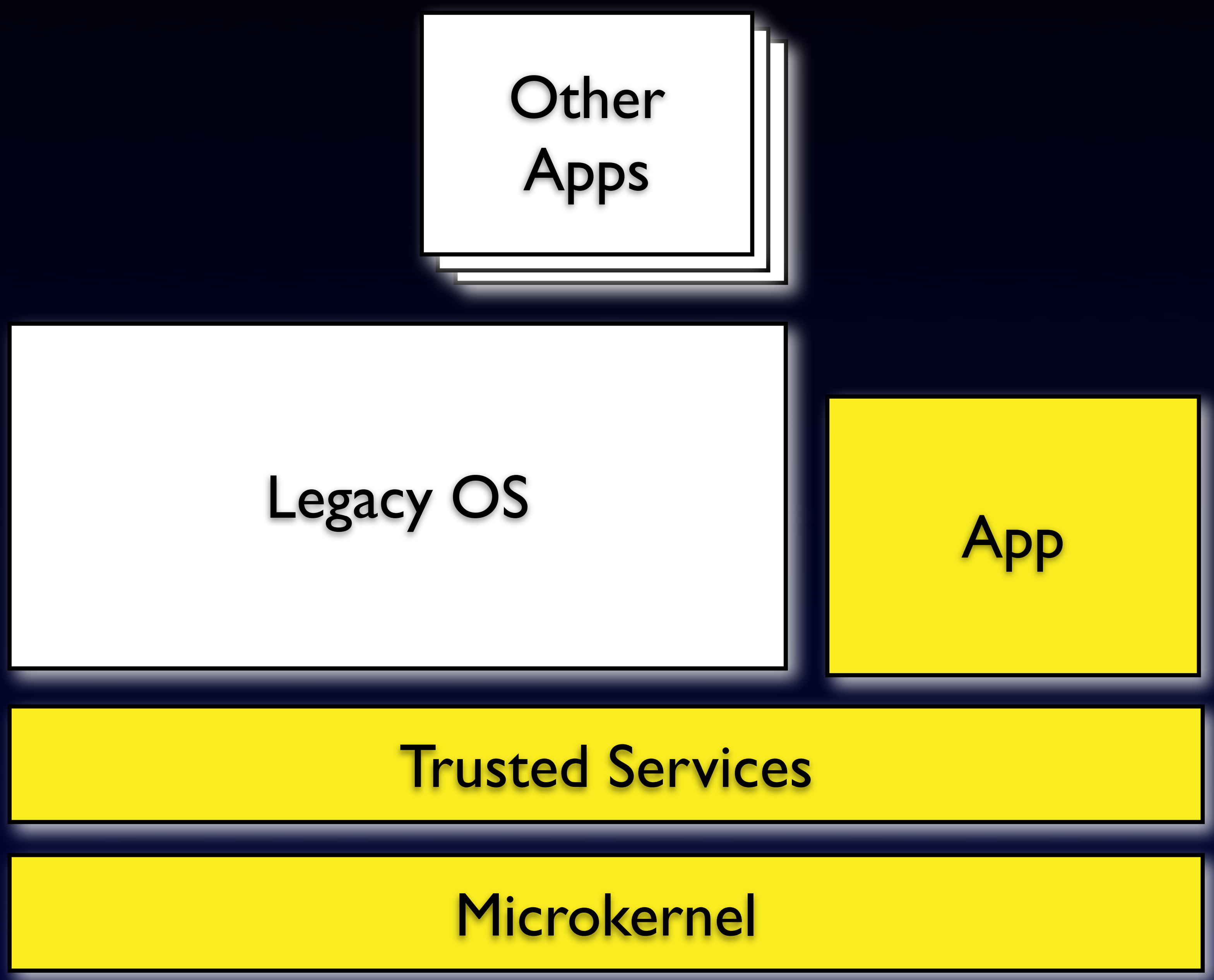
No!

Apps still *explicitly* rely on OS services, so *semantic-level attacks* are possible.

Background: Isolation Architectures



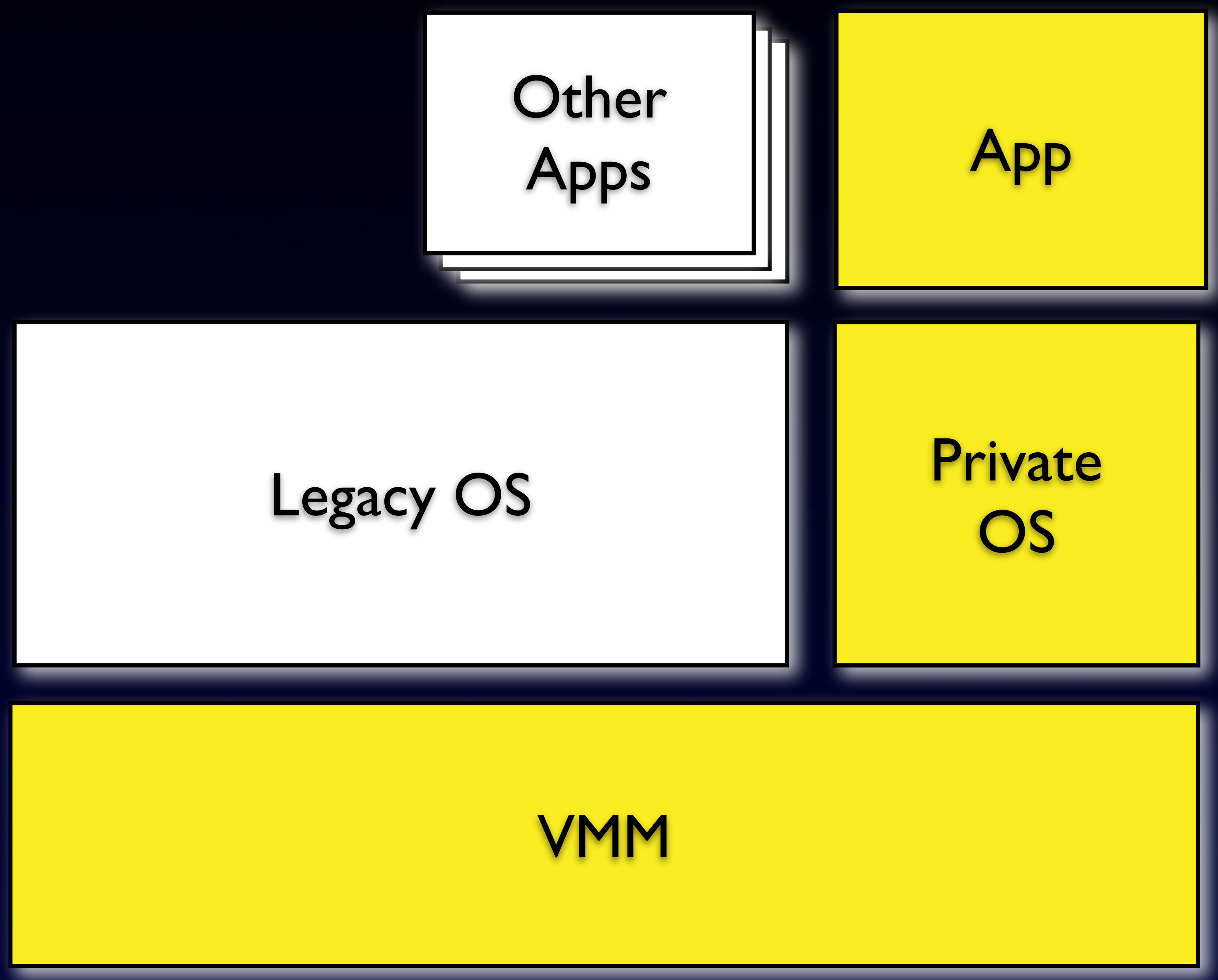
Background: Isolation Architectures



Isolation can be enforced via:

- microkernel processes (e.g. Nizza / L4)

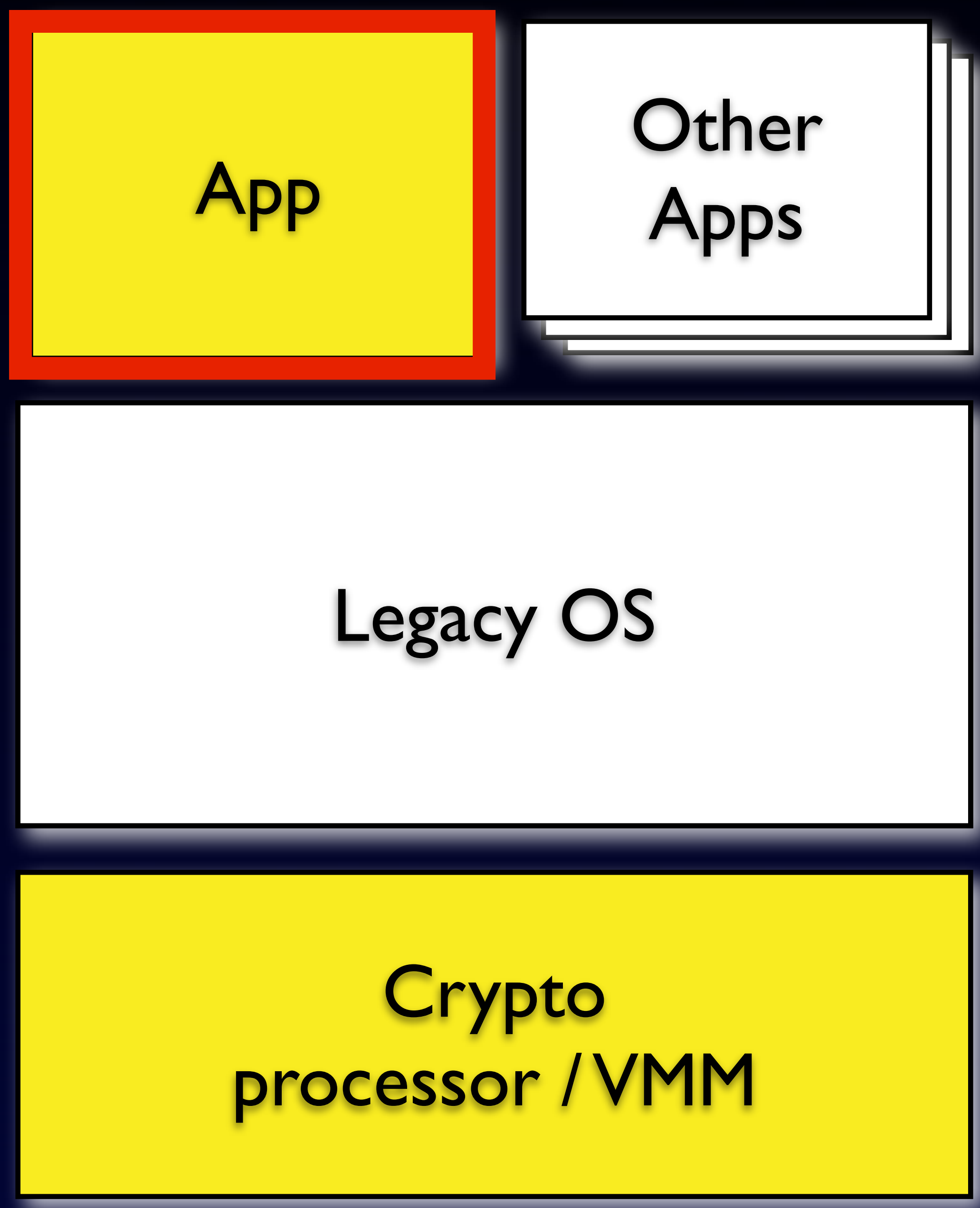
Background: Isolation Architectures



Isolation can be enforced via:

- microkernel processes (e.g. Nizza / L4)
- separate VMs (e.g. Proxos, NGSCB)

Background: Isolation Architectures



Isolation can be enforced via:

- microkernel processes (e.g. Nizza / L4)
- separate VMs (e.g. Proxos, NGSCB)
- encrypted application state (e.g. XOM, Overshadow)

Isolation Properties

- secrecy: resources can’t be read by the OS
- integrity: resources can’t be modified (without being detected)
- secure control transfer: OS can’t affect control flow, except via syscalls/signals

No defense against semantic attacks!

Malicious OS Example

Thread 1

```
acquire_lock(l);
isEncrypted = true;
encrypt(data);
release_lock(l);
```

Thread 2

```
acquire_lock(l);
if (isEncrypted) {
  sendToNet(data);
}
release_lock(l);
```

OS grants lock to both threads, introducing a new race condition!

More OS Misbehavior

A malicious OS could:

- read or modify file contents
  - even if encrypted, swap two files
- snoop on keyboard/display I/O
- change system clock (break time-based auth)
- control /dev/random (break crypto)

(more examples & solutions in paper)

Towards Application Security

Ensure that system call results are valid
(safety properties only; no availability)

Three approaches:

- verify correctness of system call results
- emulate system call in trusted layer
- disallow system call / “use at own risk”

Verifying Mutexes

Create "lock-held?" flag in shared memory

- update after lock acquired & before released
- when acquiring lock, check if already held by another thread

Isn't this just re-implementing locking?
No — OS still handles scheduling, fairness, etc.

Verifying the File System

Similar to other FSes with untrusted storage (e.g. VPFS, TDB, Sirius)

Approach:

- encrypt and hash file contents
- store file hashes, metadata in a hash tree
- need to protect directory structure too!

Emulating System Calls

- **Clock/randomness**: implement in VMM; transform system calls to hypercalls
- **IPC**: use trusted layer to send message content; use OS signals for message notification

Conclusion

Isolation is only the first step to protecting applications from a malicious OS

Need to carefully consider implications of malicious behavior by “untrusted” components

Verifying correct behavior often simpler than implementing it, so allows smaller TCB