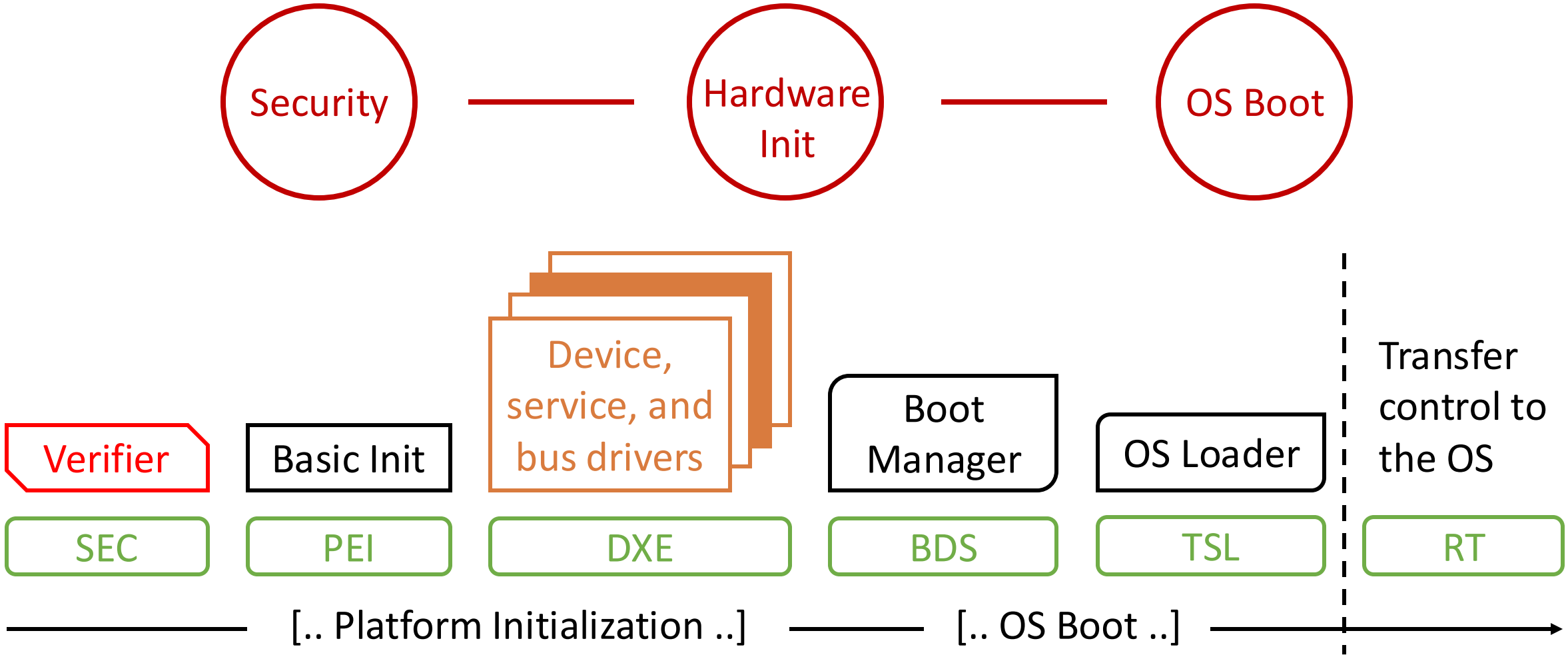
μEFI: A Microkernel-Style UEFI with Isolation and Transparency

Le Chen, Yiyang Wu, Jinyu Gu, Yubin Xia, Haibo Chen

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

IPADS
INSTITUTE OF PARALLEL
AND DISTRIBUTED SYSTEMS

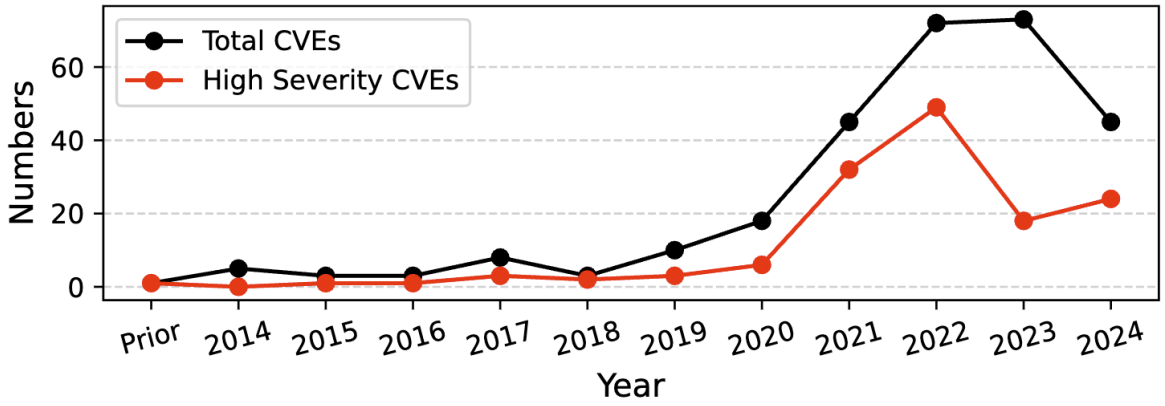
UEFI 101 – What happens before the OS boots

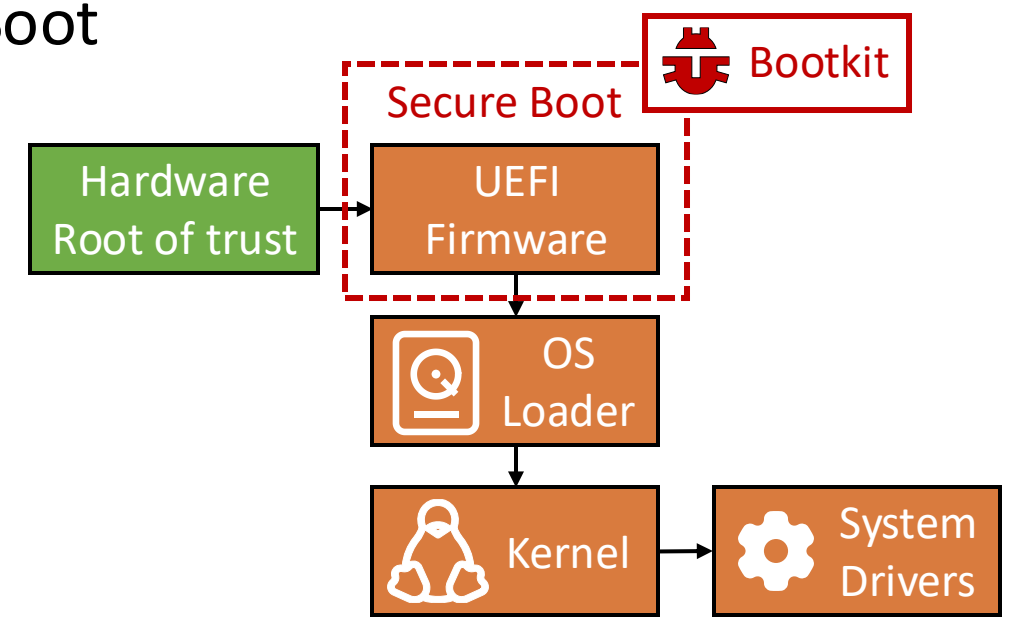


UEFI security issues

Emerging firmware attacks

- MoonBounce (2022): Capable of hiding in SPI flash memory
- BlackLotus (2022)
  - The first UEFI bootkit bypassing Secure Boot
  - Sold on hack forums for $5,000





Increase in both the number and severity of CVEs

LogoFAIL – *From splash screen to system takeover*

A real-world example of UEFI vulnerability exploitation

- Crafted logo file in ESP -> Buffer overflow in boot phase
- Data-only attack using legal modules
- Still exploitable after a year

```
ParseImage(ImageWidth, ImageWidth) {
   ...
   // Trigger an integer overflow
   OutputBuffer = AllocatePool(
     ImageWidth * ImageHeight * 2);
   ...
   idx = ImageWidth;
   // Consequent heap overflow
   OutputBuffer[idx] = a1;
}
```

Exploited to deploy the first UEFI bootkit for linux

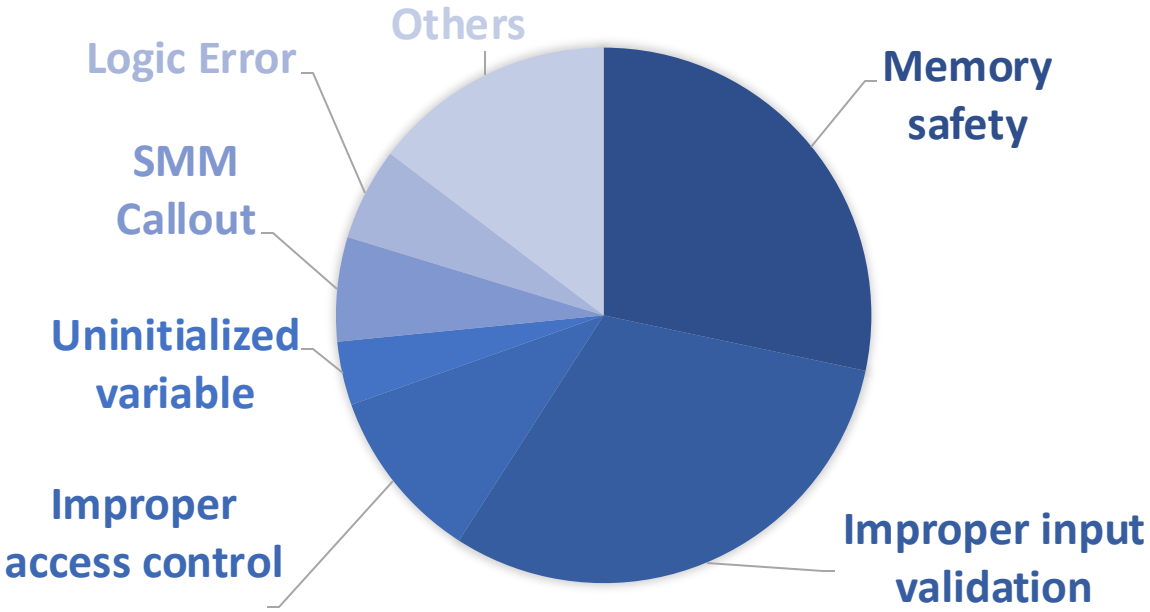
2024.11

2023.11

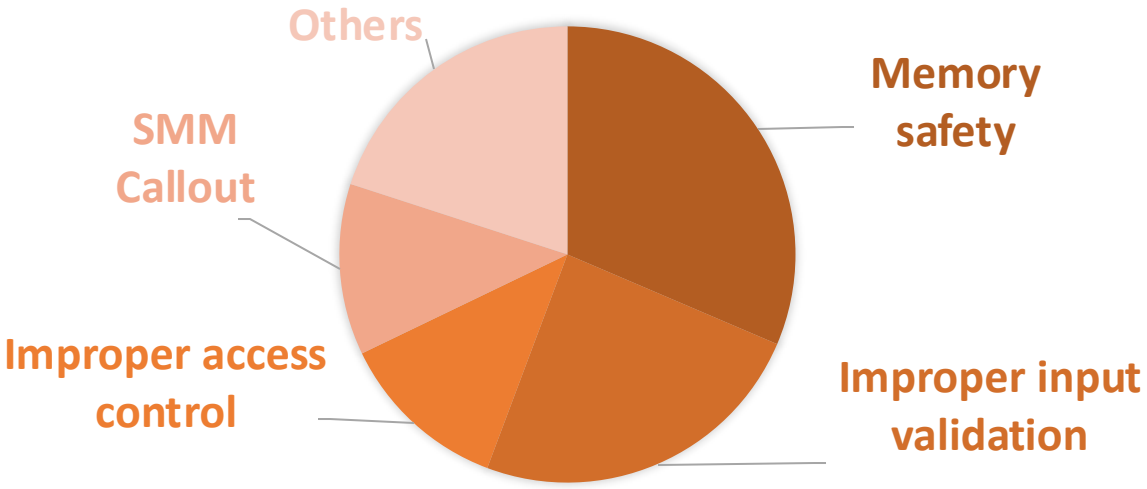
Discovered by Binarly Team

A closer look of UEFI vulnerabilities

What bugs exist?



Numbers of CVEs (286 in total)



High-severity CVEs (140 in total)

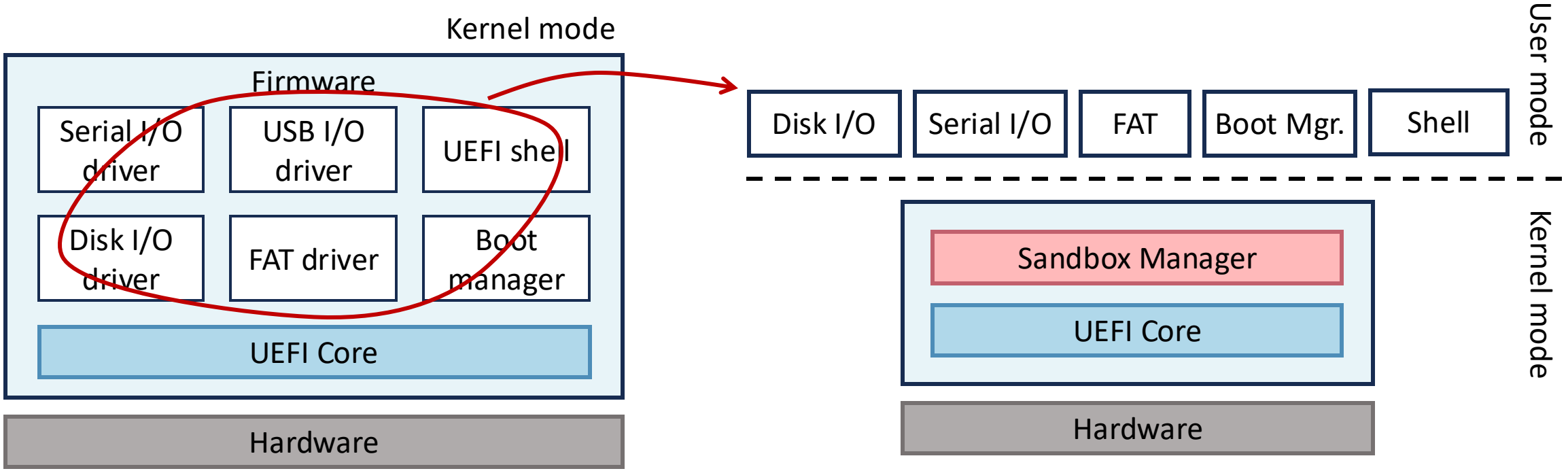
Why so damaging?

- Privileged execution environment
- Lack of inter-module isolation
- Coarse-grained access control

Idea: Revisiting microkernel in UEFI

Microkernel's philosophy: *Moving OS components into isolated user processes.*

- Aligns with the design principles of UEFI: *Modularity* & *Extensibility*
- Achieves good security and fault isolation



Requirement: Module transparency

Isolation must work with drivers we can't touch

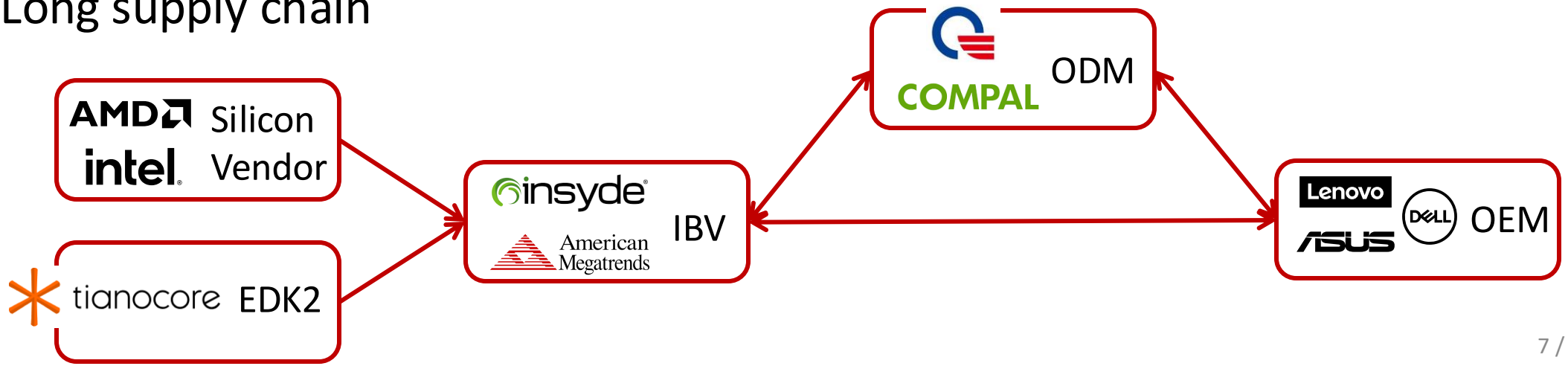
- 3rd-party DXE drivers ship without source
- Option ROMs that stored on external cards

Driver binary

Driver source code

Modifying existing modules is hard

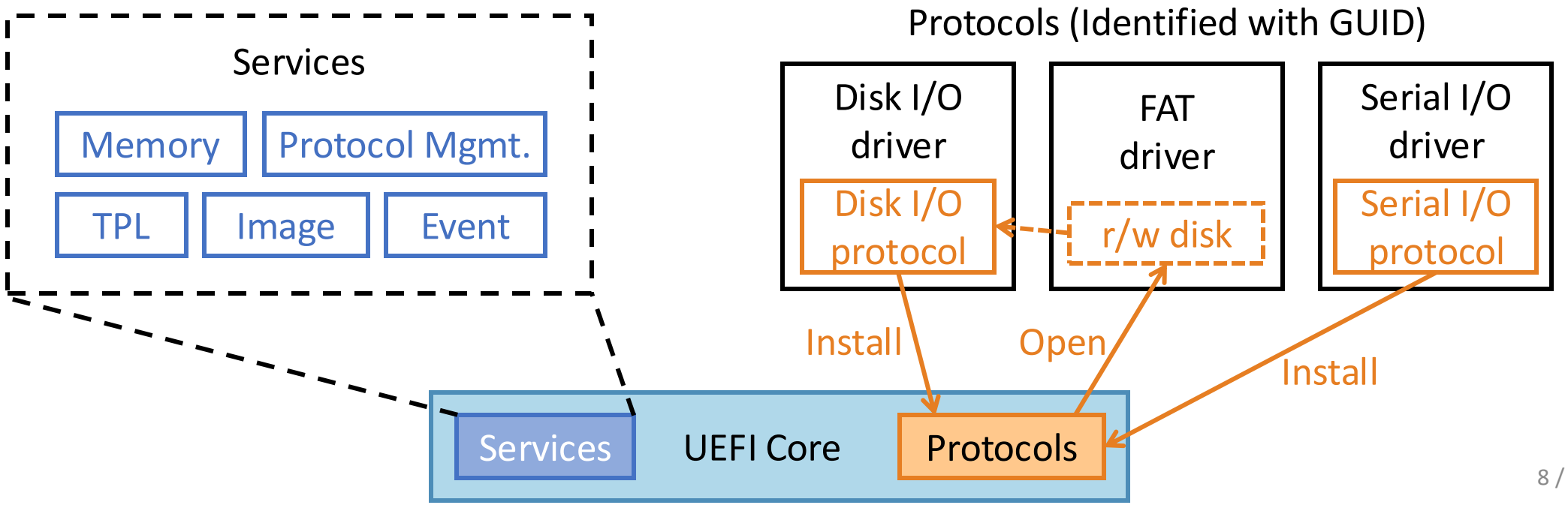
- Large number of existing modules
- Long supply chain



Challenges of module transparency

#1. Transparent control flow transfer

- **Services** and **Protocol** functions are accessed via function pointers
- Function call -> Inter-process communication (IPC)
- Protocols are runtime-discovered -> Cannot be statically rewritten

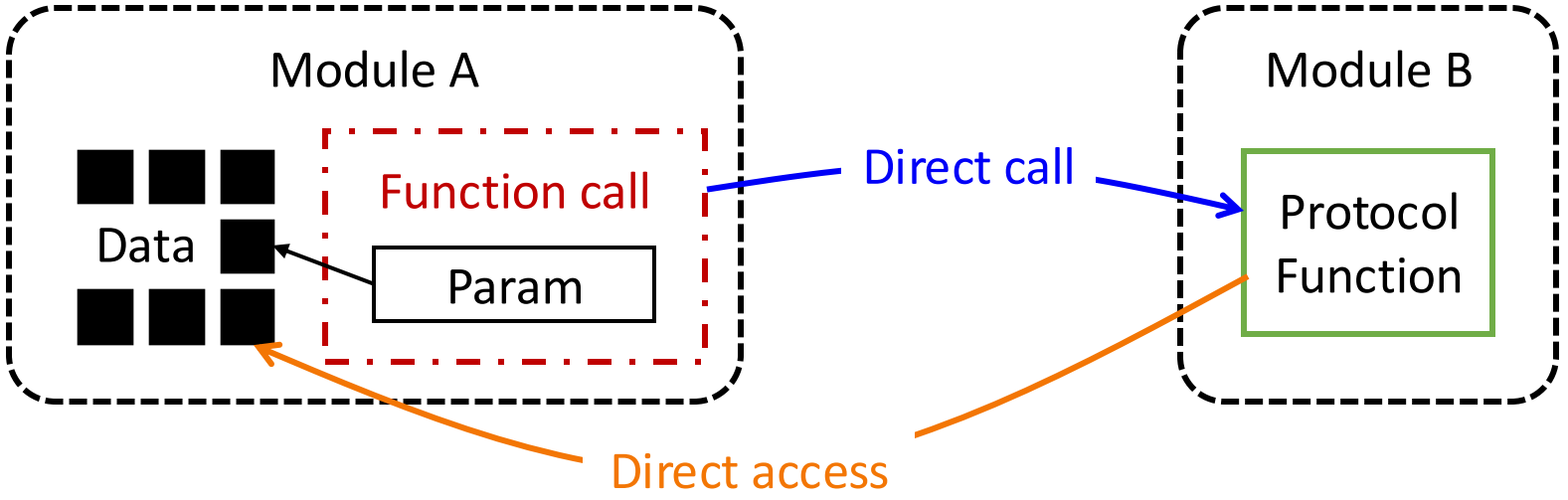


Challenges of module transparency

#2. Transparent data synchronization

– UEFI modules assume unconditional data sharing

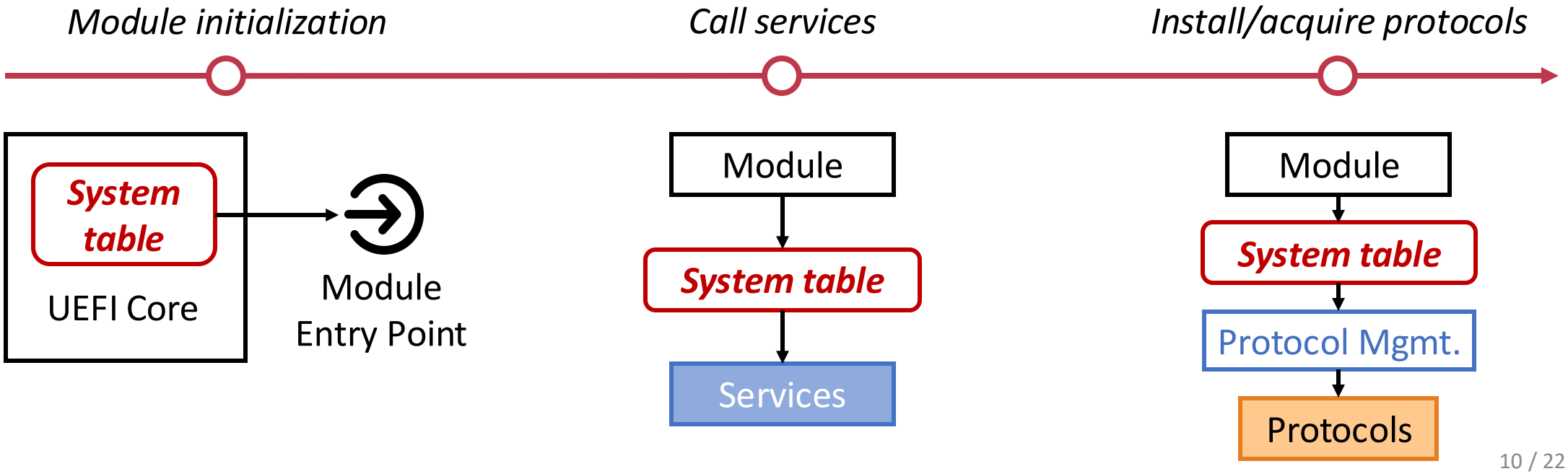
– Unified address space -> Separate address space



Observation: Unique callout gate

A central data structure: System table

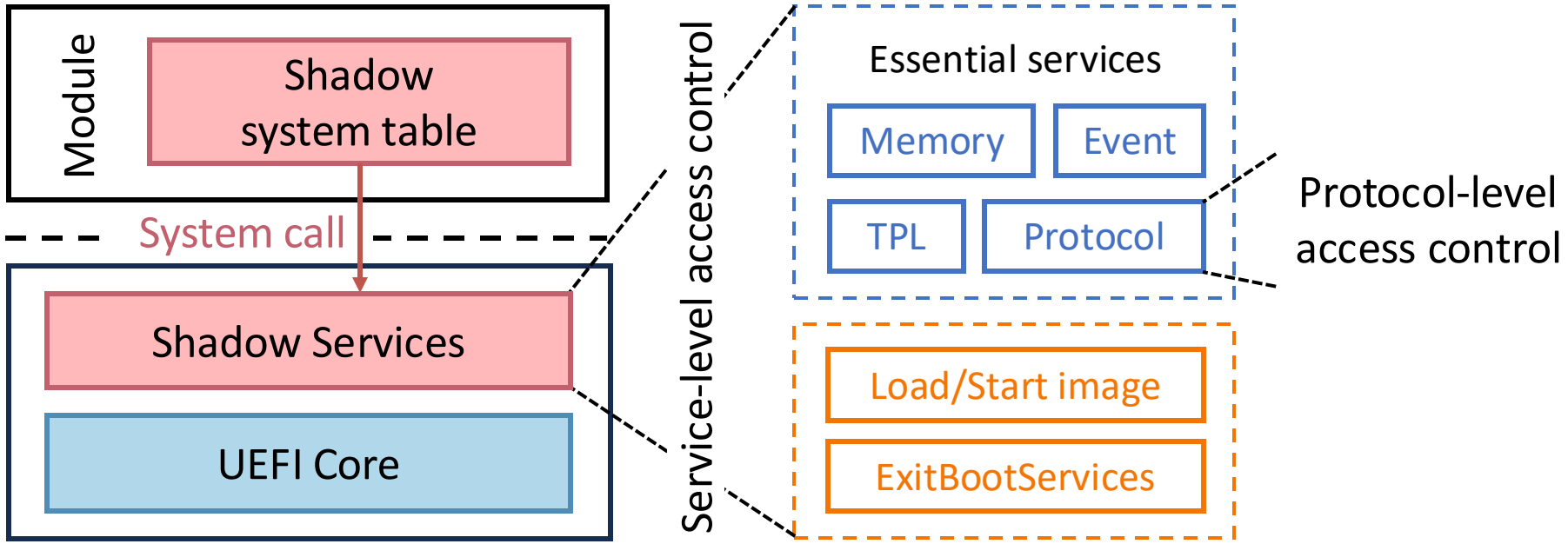
- Passed in during initialization
- Contains all the services
- Protocol interfaces are installed/acquired through services



How μEFI controls sandbox callouts

Pass the shadow system table during module initialization

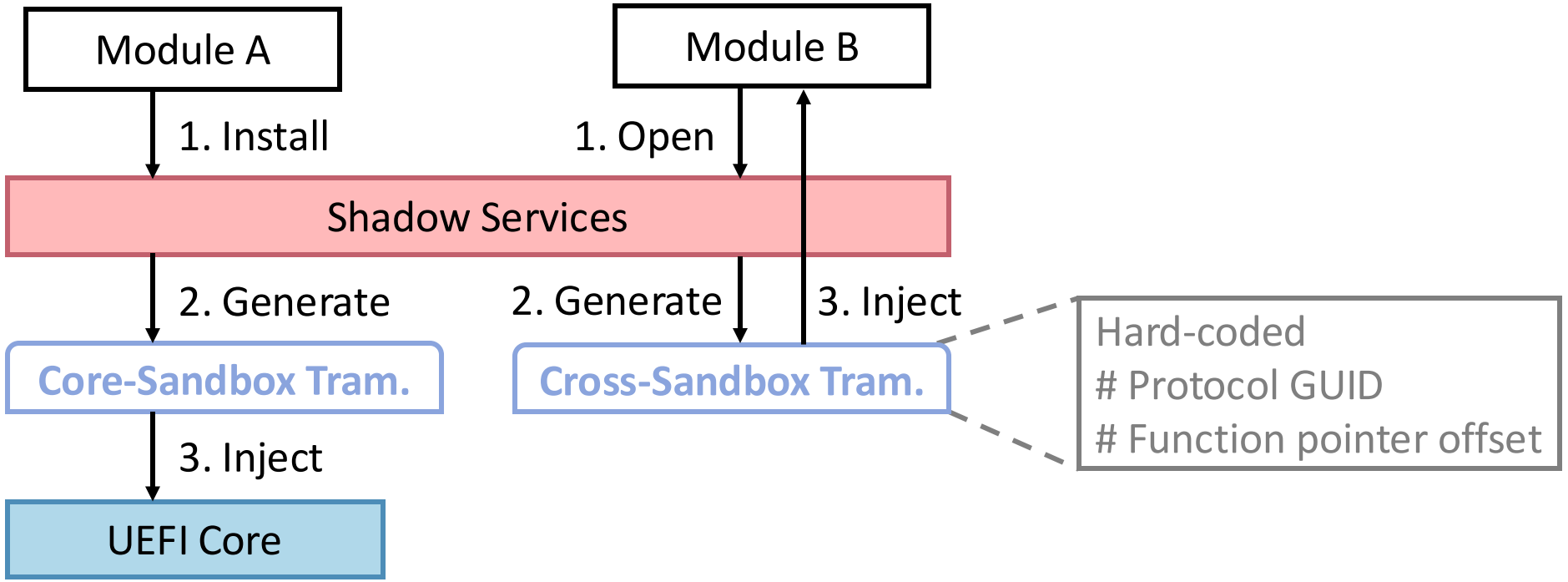
- Dynamic management
- Enforce seccomp-like access control



How μEFI handles protocol interfaces

On-the-fly trampolines for every protocol function

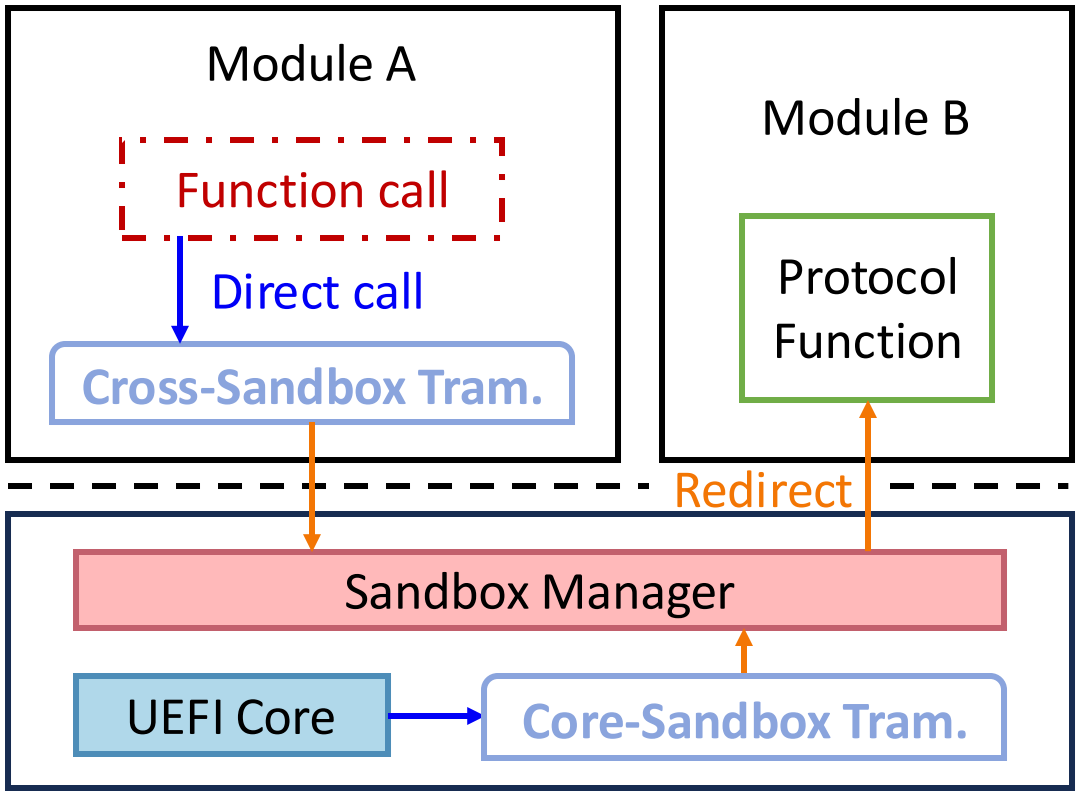
– Dynamically generated and injected at installation/lookup



How μEFI handles protocol interfaces

On-the-fly trampolines for every protocol function

– Save arguments and jump to Sandbox Manager – *No binary rewrite*



How to synchronize data?

- Data synchronization has been extensively studied in OS driver isolation
  - LXFI [SOSP '11] & KSplit [OSDI '22]

How UEFI differs?

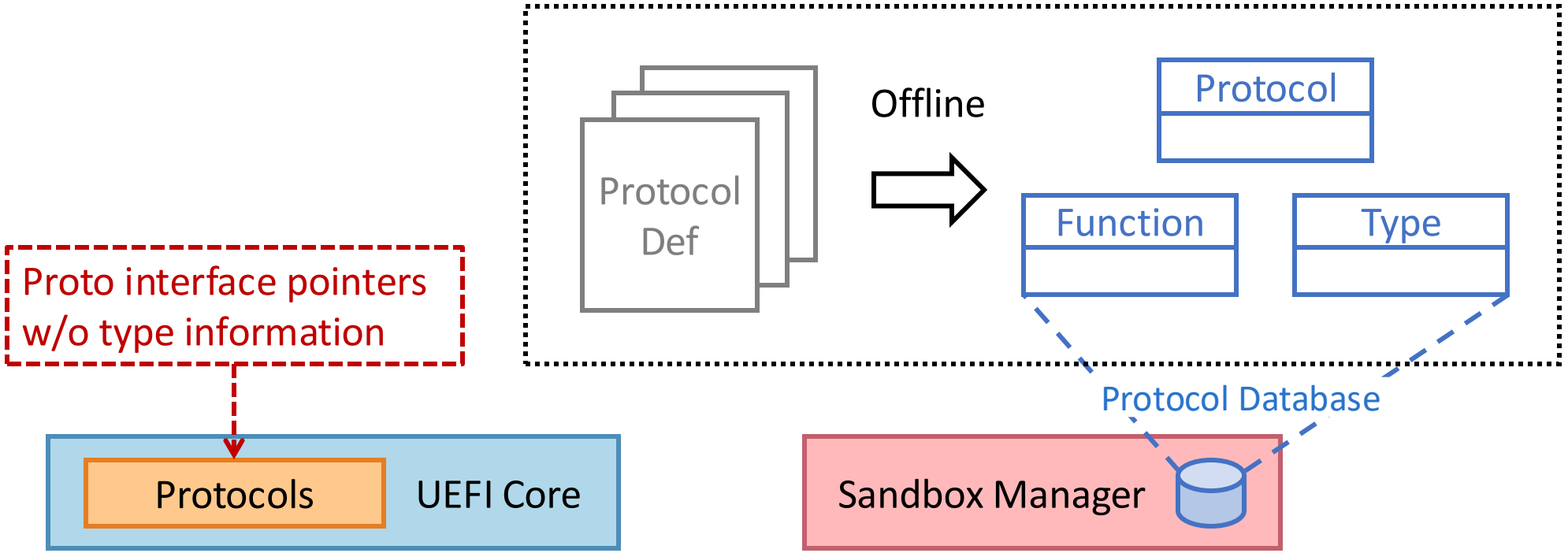
- Single-threaded execution
- Well-specified parameter scheme
  - Clear definition hide implementation details
  - IN/OUT indicators
  - Modules adhere to standard protocols

Data transfer without analyzing driver source code or binary

Step 0: Retaining type knowledge

Offline analysis of protocol headers

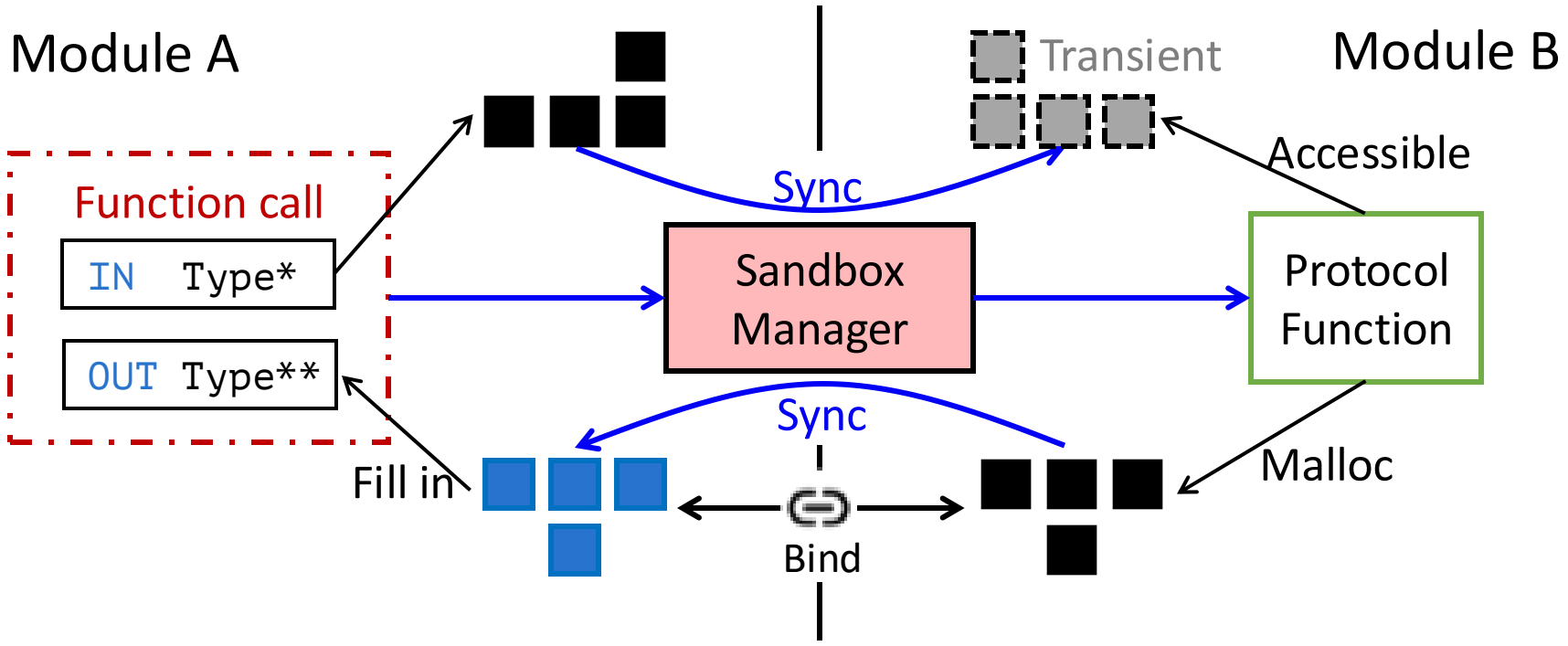
- Extract function prototypes and struct layouts
- Store into three tables: Protocol, function, type



How μEFI synchronizes data

Cross-isolation data transfer

- What to sync: IN/OUT pointers
- When to sync: Entry/Exit of protocol functions



How μEFI synchronizes data

How to interpret a pointer?

– Singleton? Variable-length buffer? Array?

Heuristic: parameter pairing

– *Buffer* <-> *BufferSize/BufferLength*

– *Array* <-> *ArrayCount*

– Failure rate < 2.5%

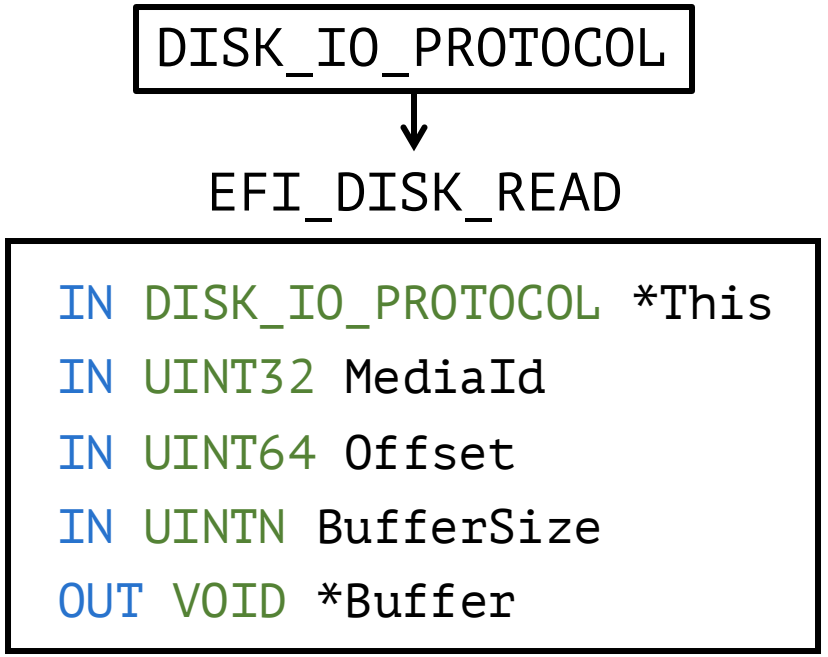
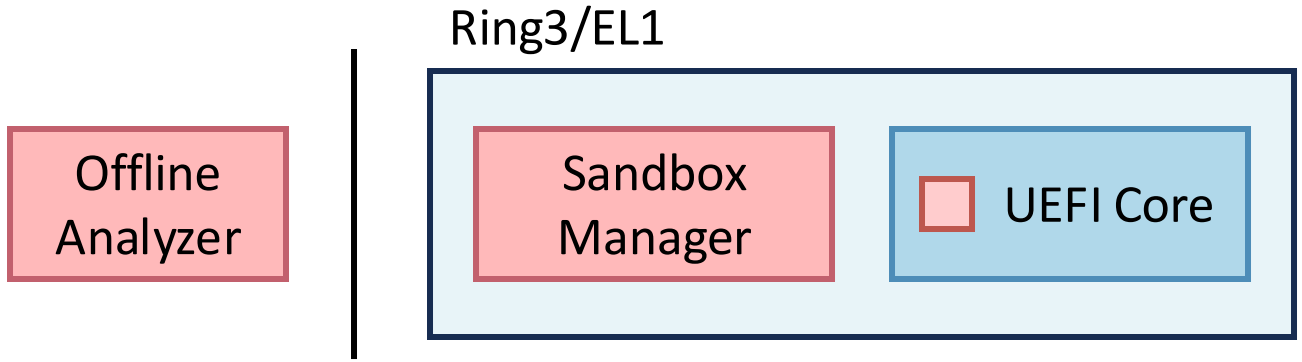


Table	Buffer w/ size	Array w/ count	Handle/ Token	Special encoding	Manual effort	Total fields
Protocol	0	0	22	10	2	1147
Type	28	1	85	189	87	33278
Function	159	6	369	101	90	3639

μEFI prototype on real-world hardware

Implementation

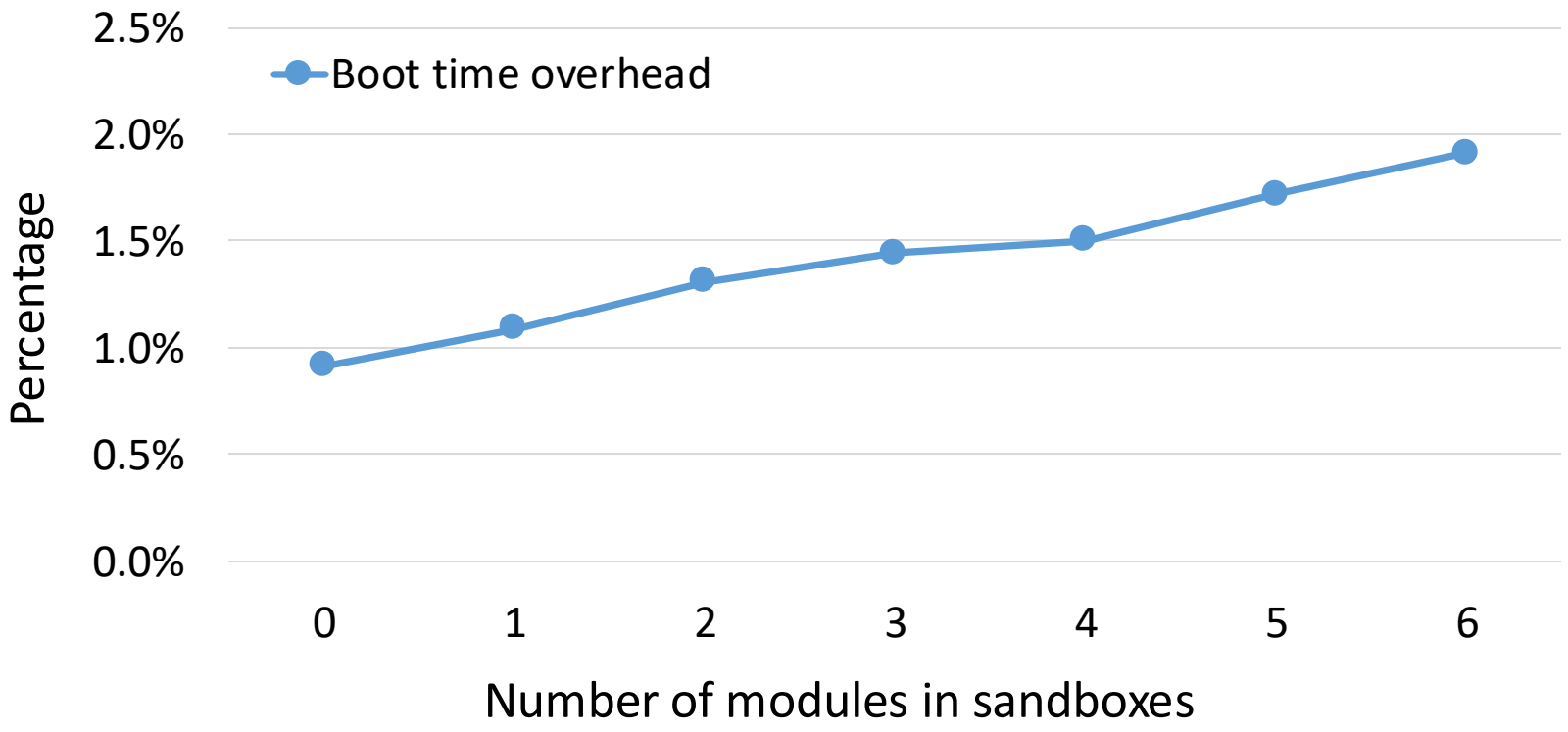
- Two platforms: QEMU-KVM (x86_64), Raspberry Pi 4 Model B (AArch64)
- Modification to the core: < 100 LoC
- Sandbox manager module: 4.7K LoC
- Offline analyzer: 900 LoC



Evaluating cost

Boot performance

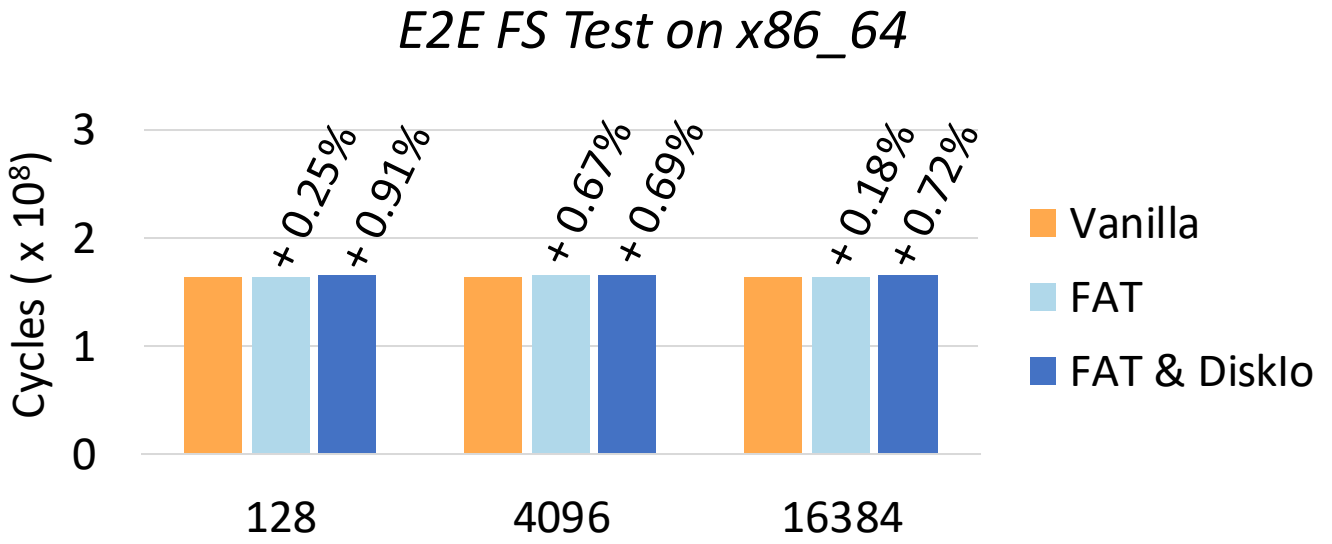
– Running six modules in sandboxes: Overhead < 2%



Evaluating cost

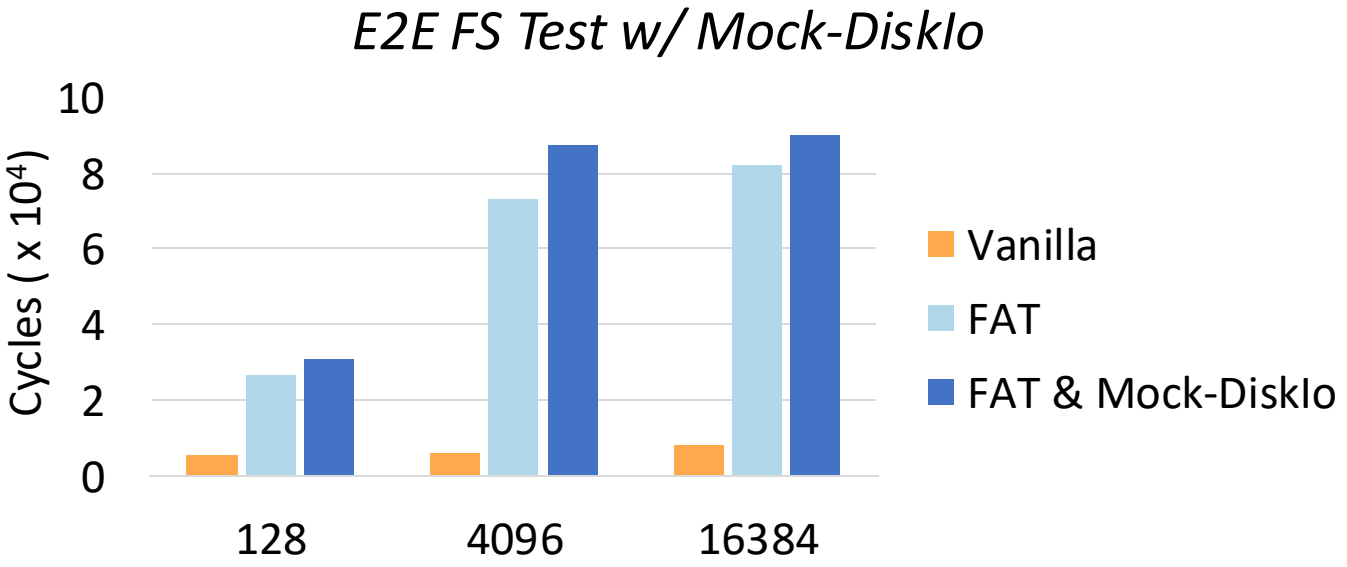
E2E FS test

- Two interactive modules
- Average overhead ~ 0.78%



FS test w/ mocked driver

- Excluding I/O operations
- Cycles: 5646 -> 26437 (FAT) -> 30870 (FAT + Mock-DiskIo)



Overhead breakdown

Examine the latency of four types of interfaces

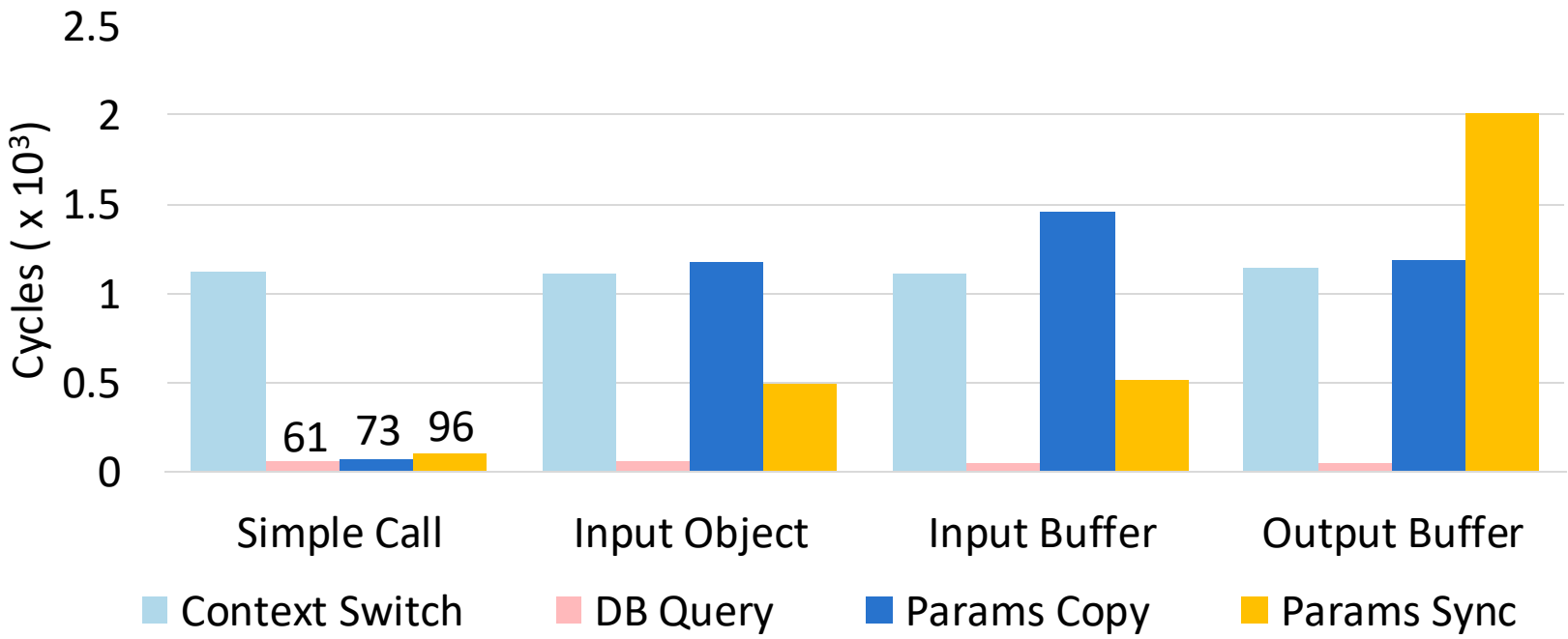
– Increases with the complexity of the interface

```
VOID
```

```
IN UINT64 *Object
```

```
IN VOID *Buffer
IN UINT64 BufferSize
```

```
OUT TYPE **OutObject
```



Summary

μEFI: A microkernel-style UEFI

- Enhanced security and fault isolation
- Seccomp-like access control

Isolation with transparency

- No need to analyze/edit driver binary/source code
- Enabled by the unique characteristics of UEFI

For questions, feel free to contact Le Chen (cen-le@sjtu.edu.cn)