# CS 477:

# Advanced Operating Systems

## Security

# This week

- OS security importance
- Balancing flexibility of security with performance using FlexOS

# OS security goals

- Traditionally: enabling multiple users to securely share a computer
  - Separation and sharing of processes, memory, files, devices, etc.

- What is the threat model?
  - Users may be malicious, users have terminal access to computers, software may be malicious/buggy, and so on

- Security mechanisms
  - Memory protection
  - Processor modes
  - User authentication
  - File access control

# OS security goals

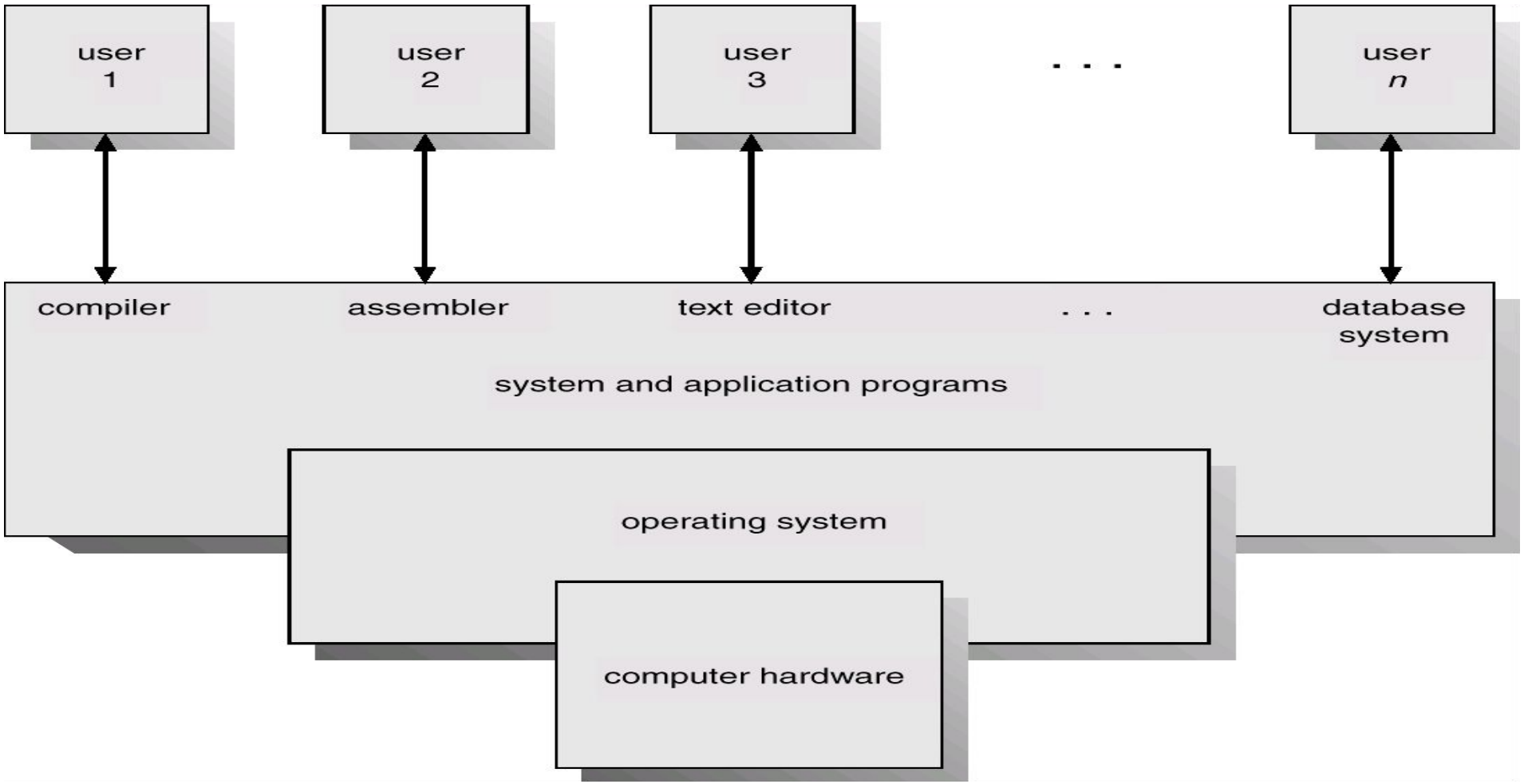
- Nowadays: ensure secure operation in networked environment
- What is the threat model?
- Security mechanisms
  - Authentication
  - Access Control
  - Secure Communication (using cryptography)
  - Logging & Auditing
  - Intrusion Prevention and Detection
  - Recovery

# Security: reconcile isolation and sharing

- Ensure separation:
  - Physical: system, data physically isolated; Rely on separate hardware
    - Prevents unauthorized access that requires physical presence to breach security
  - Temporal: Allow different time period for execution
    - Prevents interference by ensuring that only one entity has access or control at a time
  - Logical: VMs
    - Provides isolation without requiring separate physical infrastructure, making it more cost-effective and scalable
  - Cryptographical: Use encryption for ensuring data access only to authorized parties
    - Ensures data confidentiality and integrity, even if the underlying physical or logical infrastructure is shared

- OS also needs to ensure sharing

# Abstract view of system components



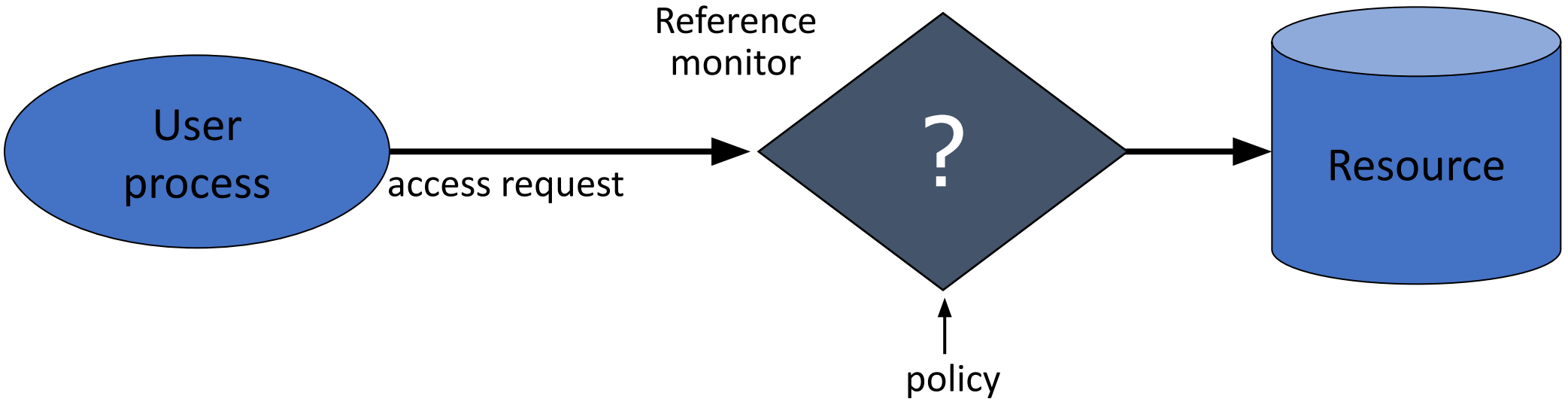
# Memory protection: access control to mem

- Ensures that one user's process cannot access other's memory
  - relocation
  - base/bounds register
  - segmentation
  - paging
  - …

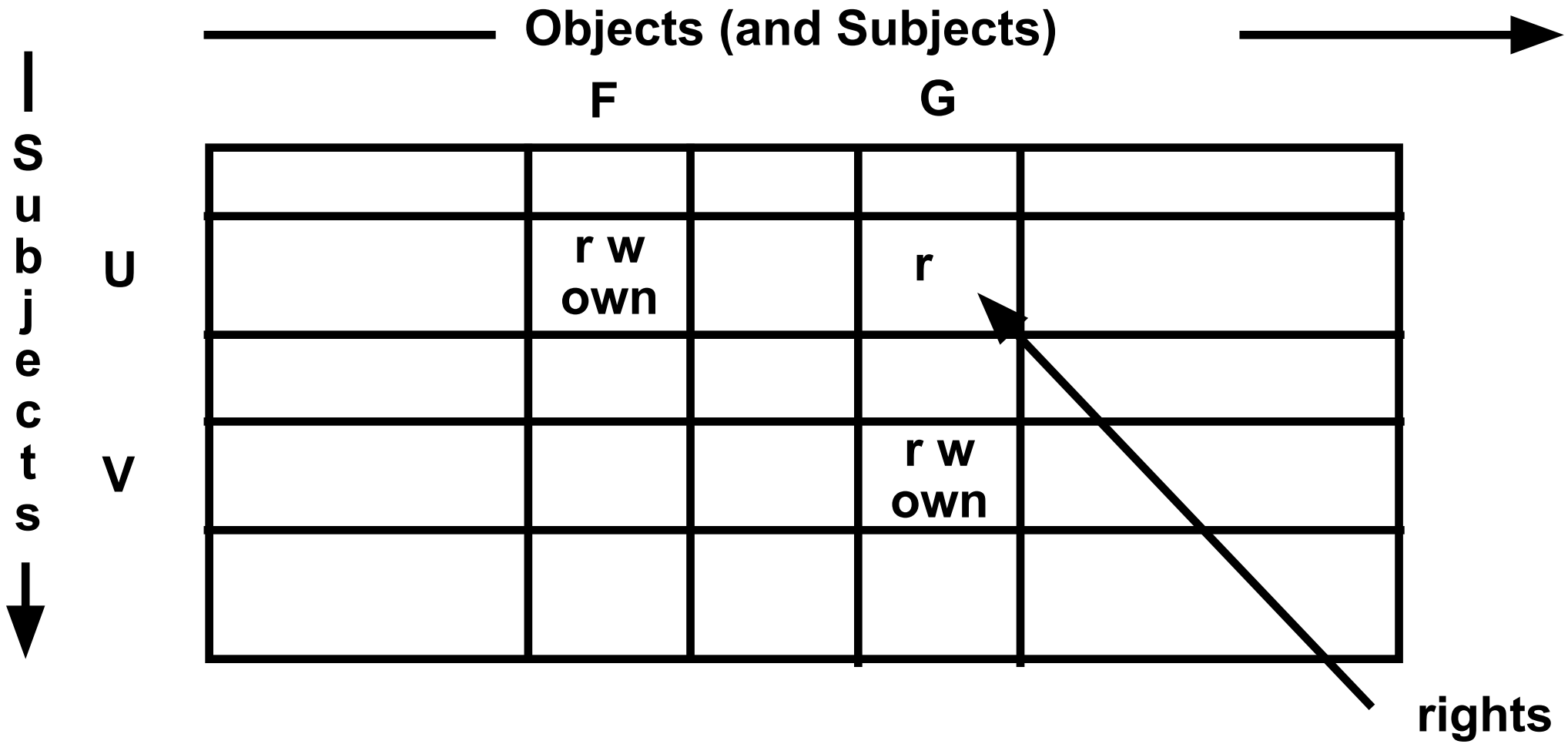
- Operating system and user processes need to have different privileges
- CPU modes: user mode vs kernel mode
  - Rely on system calls to transition from the user mode to the kernel mode

# Access control

- A **reference monitor** mediates all access to resources
  - Principle: Complete mediation: control **all** accesses to resources



# Access matrix model



# Access matrix model

- Basic abstractions:
  - Subjects
  - Objects
  - Rights
- The rights in a cell specify the access of the subject (row) to the object (column)

# Principals and subjects

A **subject** is a program (application) or an active entity that performs actions on resources in a system on behalf of a principal

A **principal** is an identifiable entity that can make requests to access resources in a system

- Principals are often associated with users, roles, or even services, applications, or systems that interact with resources.

Example:

- If “Alice” logs into a system and uses a browser to interact with a web application, Alice is the **principal**, and the browser session acting on her behalf is the **subject**

- If a service account is used by an automated script to query a database, the service account is the **principal**, and the script is the **subject**

# Unix access control: users/group/files/proc

- Each user account has a **unique ID**
- A user account belongs to multiple groups
- Subjects are processes
  - Associated with uid/gid
  - Objects are files

# Objects

- An object is anything on which a subject can perform operations (mediated by rights)
- Usually objects are passive, for example:
  - File
  - Directory (or Folder)
  - Memory segment

- But, subjects (i.e. processes) can also be objects, with operations performed on them
  - kill, suspend, resume, send interprocess communication, etc.

# Unix access control: users/group/files/proc

- Each user account has a **unique ID**
- A user account belongs to multiple groups
- Subjects are processes
  - Associated with uid/gid
  - Objects are files

# Object organization

- Almost all objects are organized as files
  - Files are arranged in hierarchy
  - Files exist in directory
  - Directories are also one type of file
- Each object has
  - Owner
  - Group
  - 12 permission bits
    - RWX for owner, group, and others (3x3)
    - suid, sgid, sticky

# Basic permission bits on files (non-directories)

- Read controls reading the content of a file
  - i.e., the read system call

- Write controls changing the content of a file
  - i.e., the write system call

- Execute controls loading the file in memory and execute
  - i.e., the execve system call

# Permission bits on directories

- Read bit allows one to show file names in a directory
- The execution bit controls traversing a directory
  - does a lookup, allows one to find inode # from file name
  - chdir to a directory requires execution

- Write + execution control creating/deleting files in the directory
  - Deleting a file under a directory requires no permission on the file

- Accessing a file identified by a path name requires execution to all directories along the path

# The three sets of permission bits

- Intuition:
  - if the user is the owner of a file, then the r/w/x bits for owner apply
  - Otherwise, if the user belongs to the group the file belongs to, then the r/w/x bits for group apply
  - Otherwise, the r/w/x bits for others apply

- Can one implement negative authorization, i.e., only members of a particular group are not allowed to access a file?

# Other issues on objects in UNIX

- Accesses other than read/write/execute
  - Who can change the permission bits?
    - The owner can
  - Who can change the owner?
    - Only the superuser

- Rights not related to a file
  - Affecting another process
  - Operations such as shutting down the system, mounting a new file system, listening on a low port
    - Traditionally reserved for the root user

# Process user ID model in Modern UNIX

- Each process has three user IDs
  - real user ID (ruid) owner of the process
  - effective user ID (euid) used in most access control decisions
  - saved user ID (suid) stores the original elevated privileges

- And three group IDs
  - real group ID
  - effective group ID
  - saved group ID

# Process user ID model in Modern UNIX

- When a process is created by *fork*
  - it inherits all three users IDs from its parent process

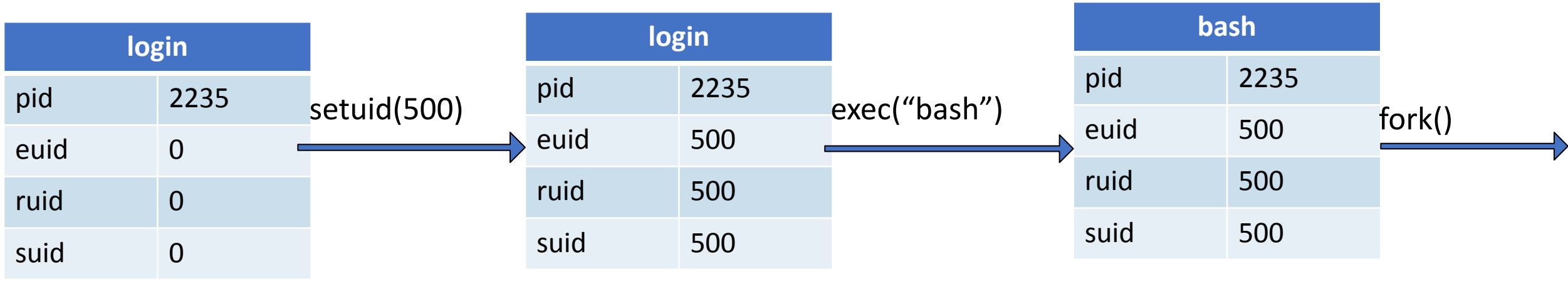
- When a process executes a file by *exec*
  - it keeps its three user IDs unless the set-user-ID bit of the file is set, in which case the effective uid and saved uid are assigned the user ID of the owner of the file

- A process may change the user IDs via system calls

# Needing suid/sgid bits

- Some operations are not modeled as files and require user id = 0
  - halting the system
  - bind/listen on “privileged ports” (TCP/UDP ports below 1024)
  - non-root users need these privileges

# Events during logging




After the login process verifies that the entered password is correct, it issues a setuid system call.

The login process then loads the shell, giving the user a login shell.

The user types in the passwd command to change his password.

| bash | |
|---|---|
| pid | 2235 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

| bash | |
|---|---|
| pid | 2297 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

exec("passwd")

| passwd | |
|---|---|
| pid | 2297 |
| euid | 0 |
| ruid | 500 |
| suid | 0 |

Drop privilege permanently

| passwd | |
|---|---|
| pid | 2297 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

Drop privilege temporarily

| passwd | |
|---|---|
| pid | 2297 |
| euid | 500 |
| ruid | 500 |
| suid | 0 |

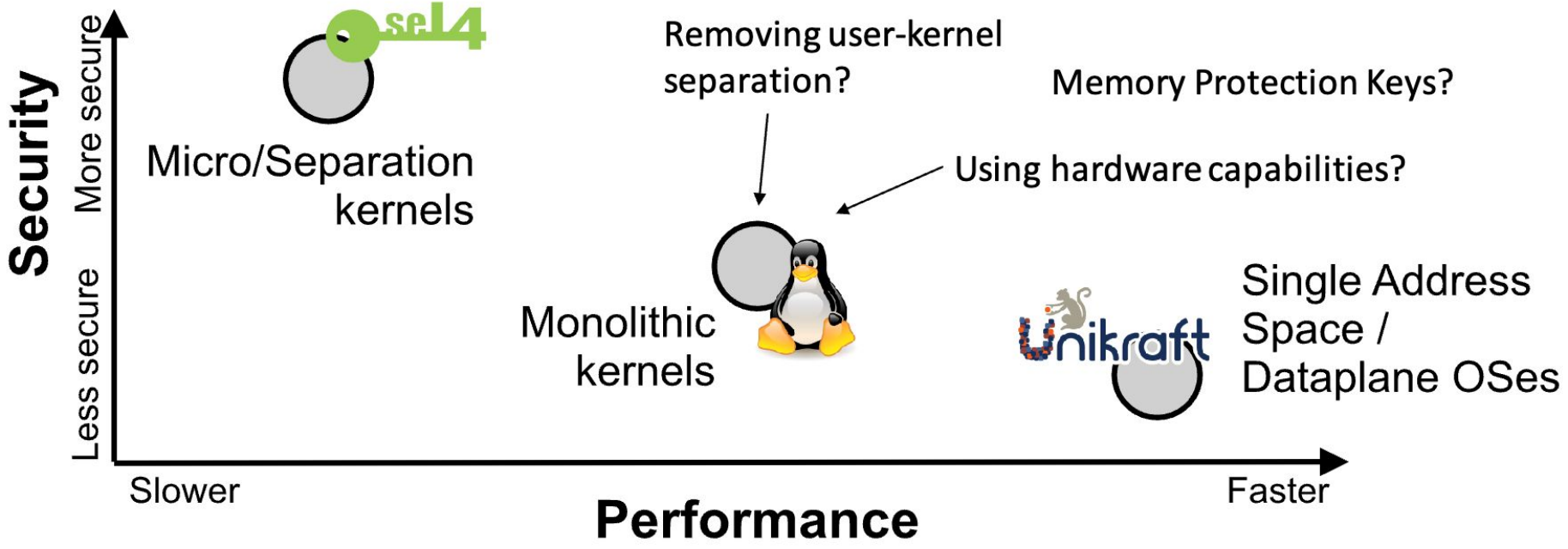
The fork call creates a new process, which loads "passwd", which is owned by root user, and has setuid bit set.

# This week

- OS security importance
- **Balancing flexibility of security with performance using FlexOS**

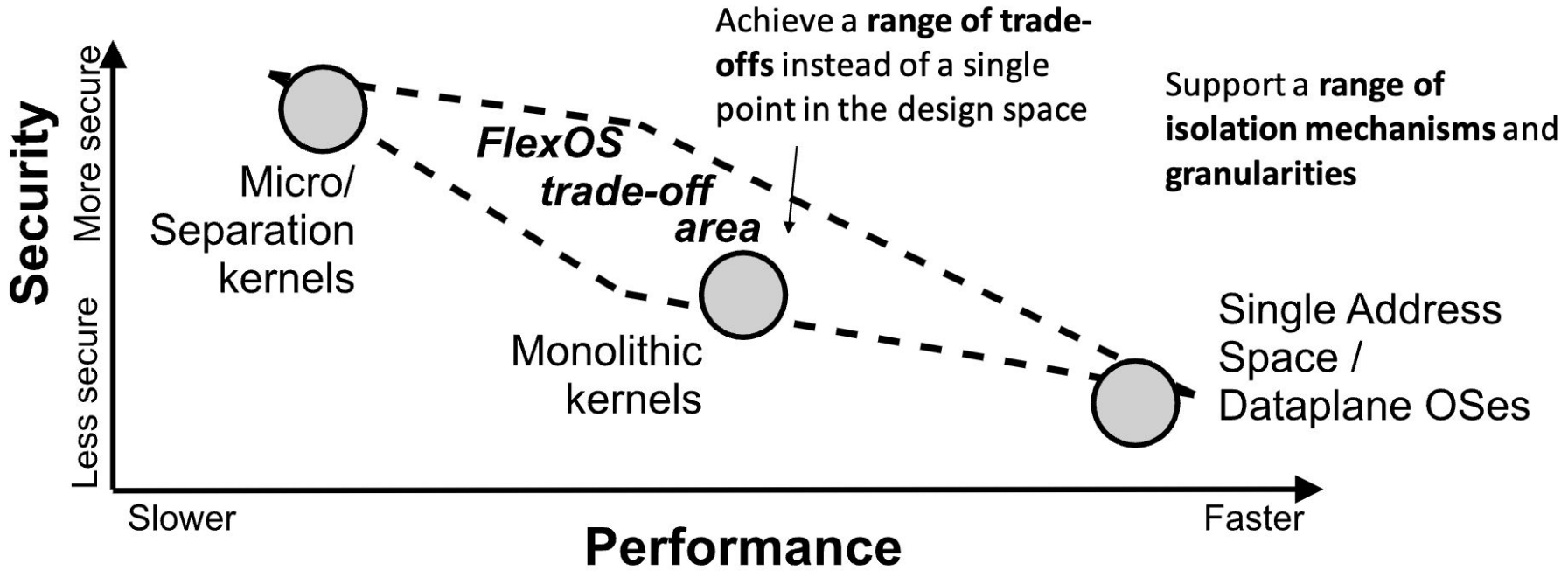
# OS designs have fixed security definitions

- OS security/isolation strategies are fixed at design time!
  - Isolation granularity, underlying mechanisms, data sharing strategies (copy/share)



# FlexOS: flexible OS isolation

- **Decouple security/isolation** decisions from the **OS design**



# Other use cases for flexible OS isolation

- Deployment to heterogeneous hardware
  - Make optimal use of each machine/architecture's safety mechanisms with the same code
- Quickly isolate vulnerable libraries
  - React easily and quickly to newly published vulnerabilities while waiting for a full patch
- Incremental verification of code-bases
  - Mix and match verified and non-verified code-bases while preserving guarantees

# Realizing FlexOS

1. Focus on **single-purpose applications** such as cloud microservices

   Running more applications leads to lesser specialization

2. **Full-system** understanding of compartmentalization

   Consider every component, i.e., both kernel and the application

   Embrace the **library OS** philosophy, everything is a library

3. **Abstract away** the technical details of isolation mechanisms

   Page table, MPK, Cheri, TEE? One can design similar interface for them

4. **Flexibility** must not get into the **way of performance**

# Overview of FlexOS

Happy users

config.yaml

① Input config

*FlexOS Toolchain*

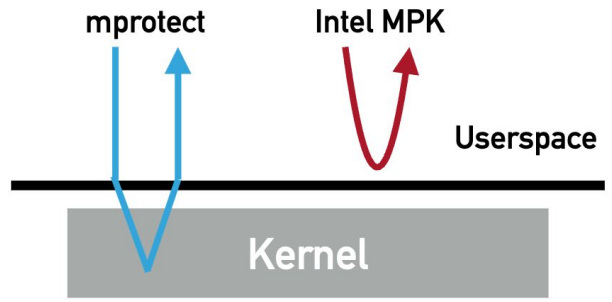
config.yaml

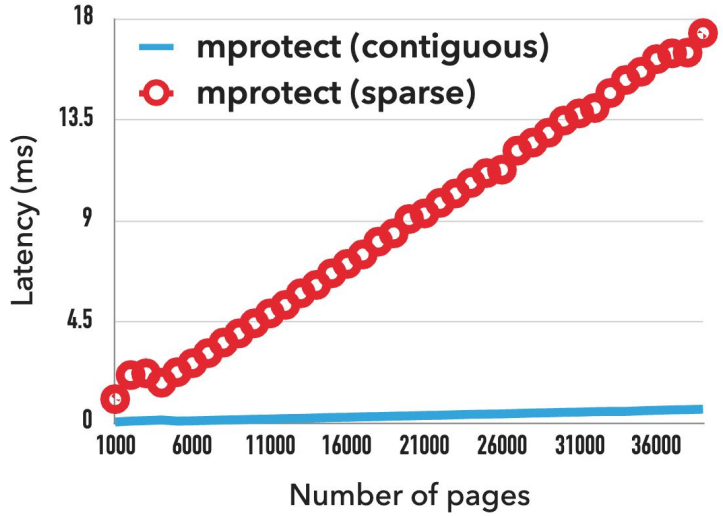
```
compartments:
- comp1:
  mechanism: intel-mpk
  default: True
- comp2:
   mechanism: intel-mpk
  hardening: [cfi, asan]
libraries:
- libredis: comp1
- libopenjpg: comp2
- lwip: comp2
```

*"Redis image with two compartments, isolate libopenjpeg and lwip together"*

# MPK primer

- Support fast permission change for page groups with single instruction
  - Fast single invocation; Fast permission change for multiple pages
  - Tags pages with PKEY
  - Permission register (PKRU): 16 MPK regions
  - Provides fast read/write permission change
    - 11-260 cycles



# Overview of FlexOS

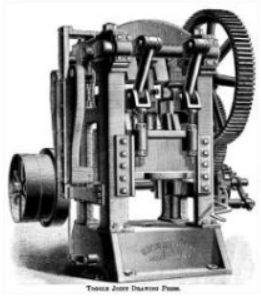
*Happy users*

config.yaml

① Input config

*Isolation Backends*

MPK | EPT | ...



② Select isolation mechanism ("Backend")

*Core Libraries*

boot | sched | mm

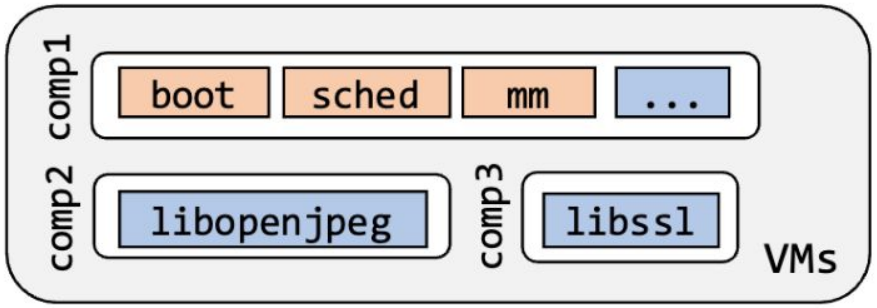
*Kernel & User Libs*

ramfs | vfs | ...

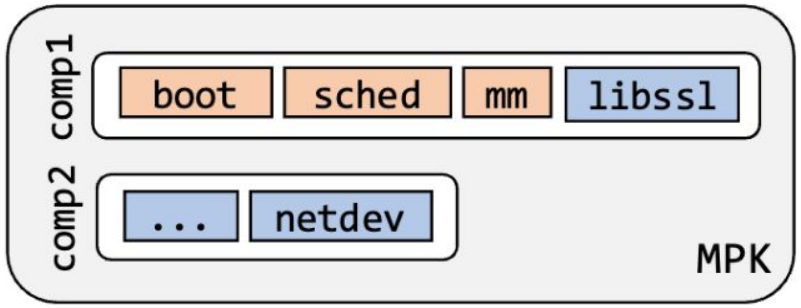
③ Select libraries (kernel and app), *rewrite*, and statically put in compartments

*FlexOS Toolchain*

④ Generate image with appropriate isolation properties




Possible Image 1




Possible Image 2

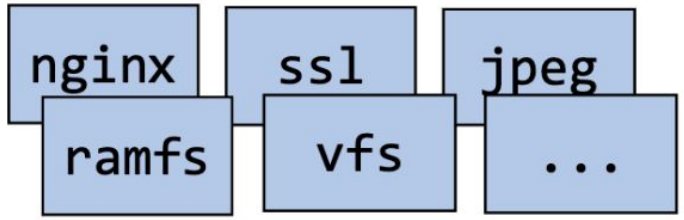
# Mechanism abstraction in FlexOS

Based on a **highly modular LibOS design** (Unikraft, *EuroSys'21*)


Such libOSes are composed of *fine-granular*, *independent* libraries

*Core Libraries*


*Kernel & User Libraries*



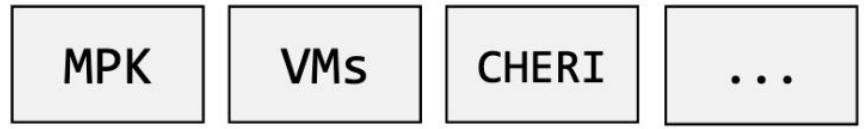

Reuse libraries as finest granularity of compartmentalization

*"Pre-compartmentalize" them*

Cross-library **calls and shared data** are replaced by an **abstract construct** (**gates**, data sharing primitives)

Defined as part of the **FlexOS API**

At build time, the toolchain replaces these constructs with particular implementations. Implementations are defined by the **backends**.



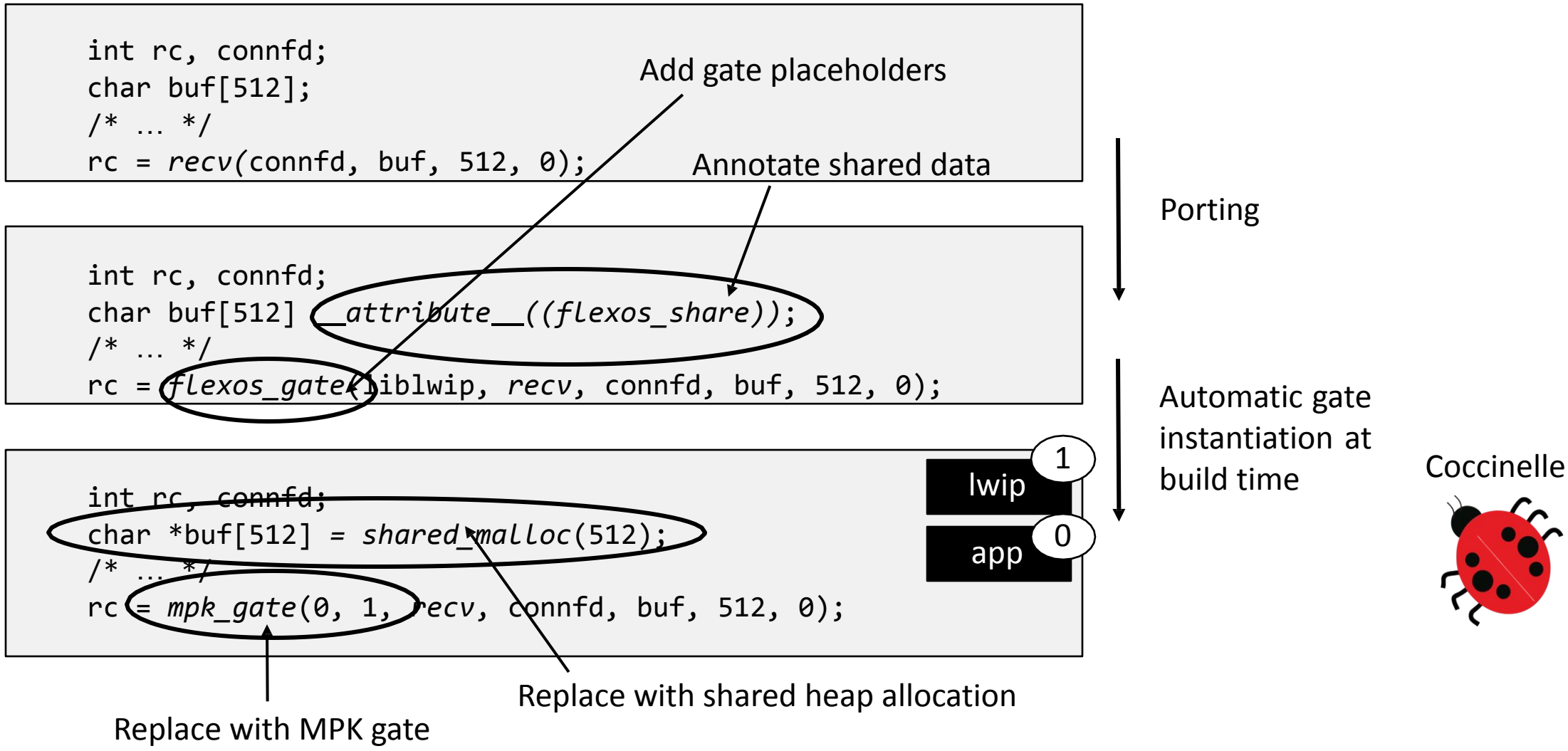
# Gate

A mechanism to controlled access to certain critical resources or operations,

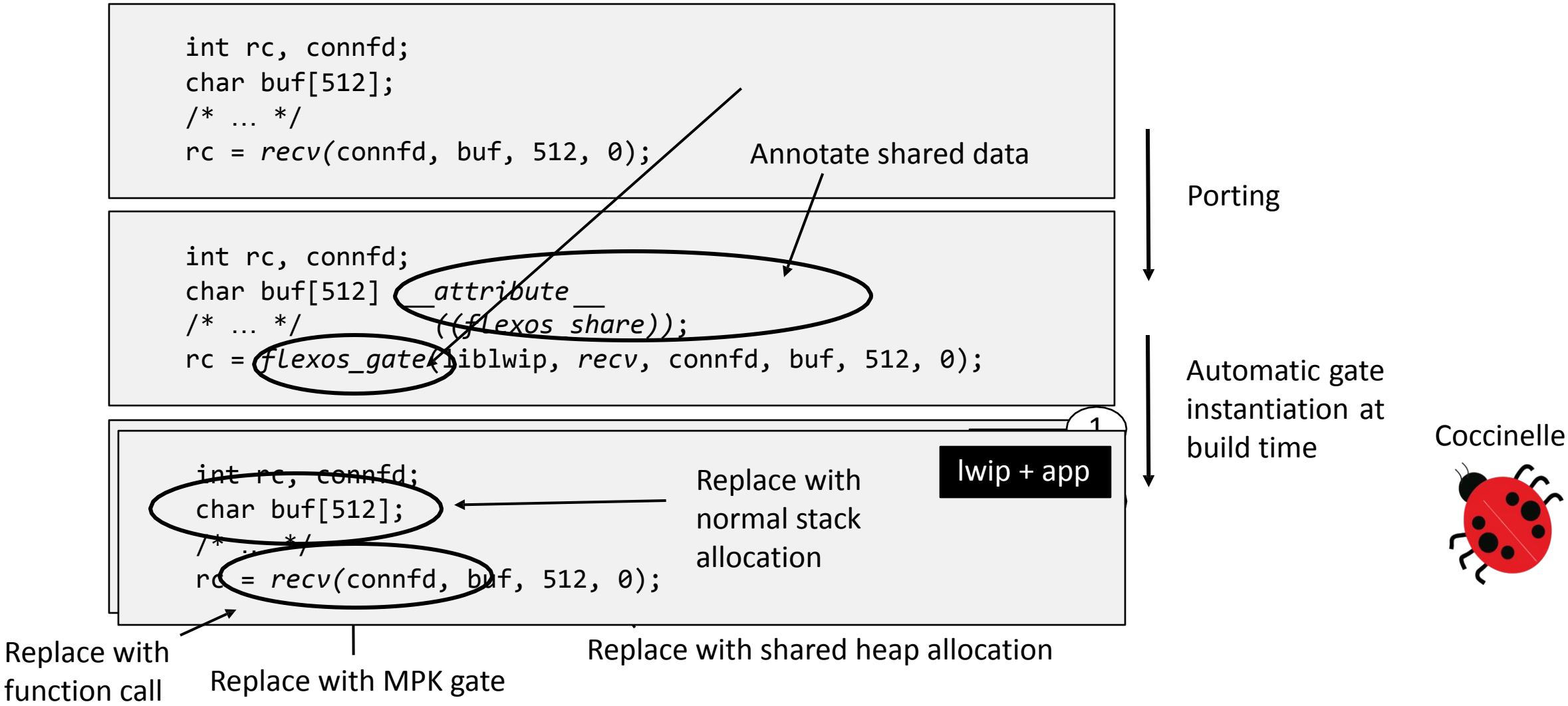
i.e., secure transitions between operations

- **Interrupt gate**: Transfers control to an interrupt handler with disabled interrupts
- **Trap gate**: Similar to interrupt gate but allows further interrupts
- **Call gate**: Allows controlled transfer of execution to a higher privilege level

# Compartmentalization toolchain



# Compartmentalization toolchain



# FlexOS prototype

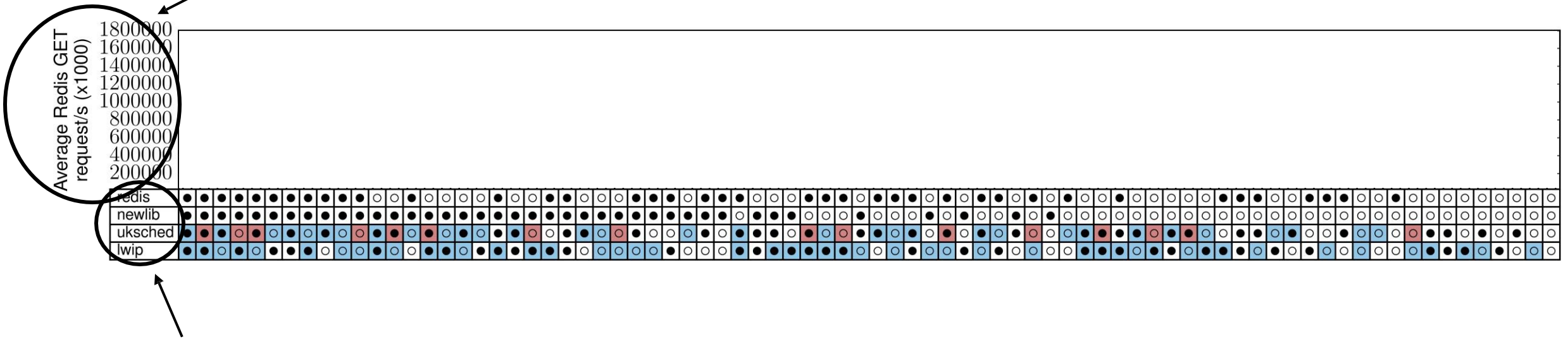
- Implemented on top of Unikraft

- Backend implementation for Intel MPK, and VMs (EPT)
  - Port of libraries: network, storage, scheduler, file system, time subsystem

- Applications: Redis, Nginx, SQLite, iPerf server

# Flexibility


Runtime performance with Redis in requests/s




FlexOS libraries used in the Redis image

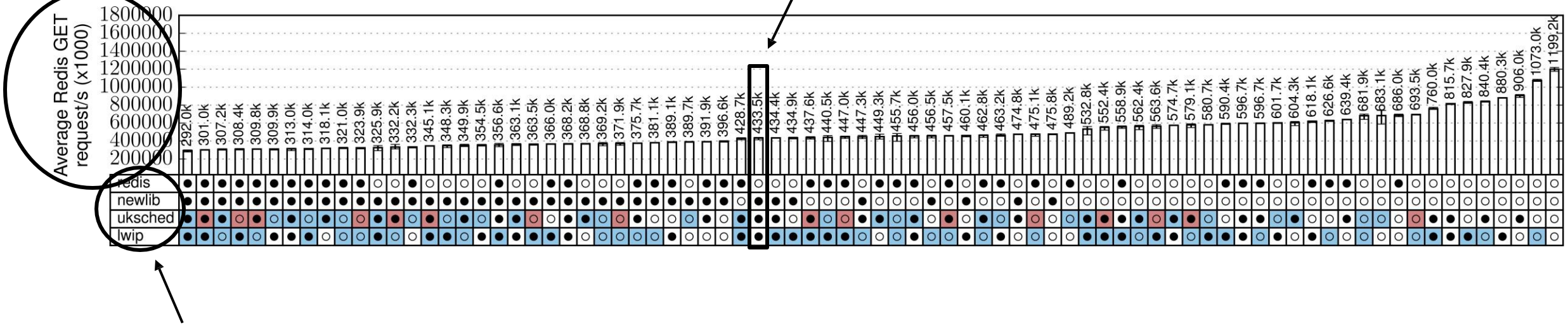
(only a subset for readability):

- Redis application
- C standard library (newlib)
- FlexOS scheduler (*uksched*)
- Network stack (lwip)

# Flexibility


Runtime performance with Redis in requests/s

One configuration and its associated performance
(80 configurations in total)




FlexOS libraries used in the Redis image

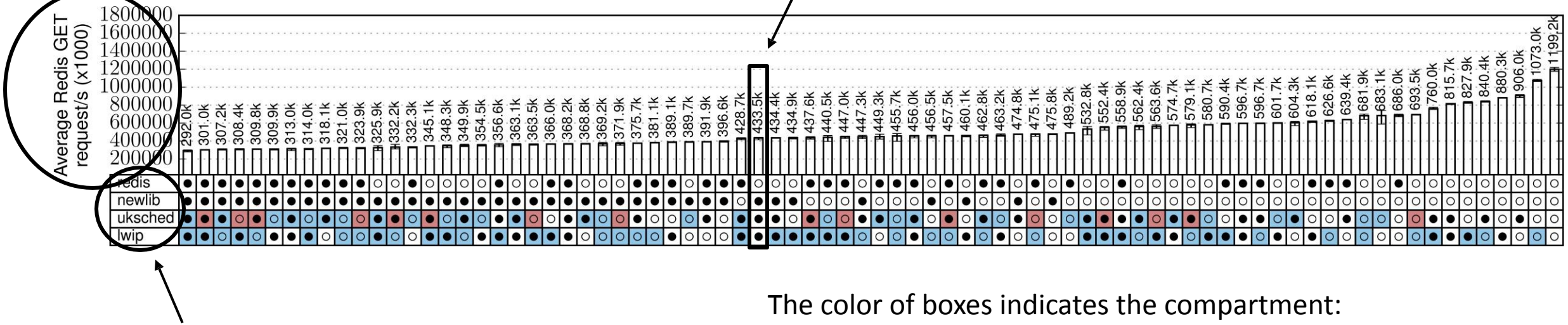
(only a subset for readability):

- Redis application
- C standard library (newlib)
- FlexOS scheduler (*uksched*)
- Network stack (lwip)

# Flexibility


Runtime performance with Redis in requests/s

One configuration and its associated performance
(80 configurations in total)



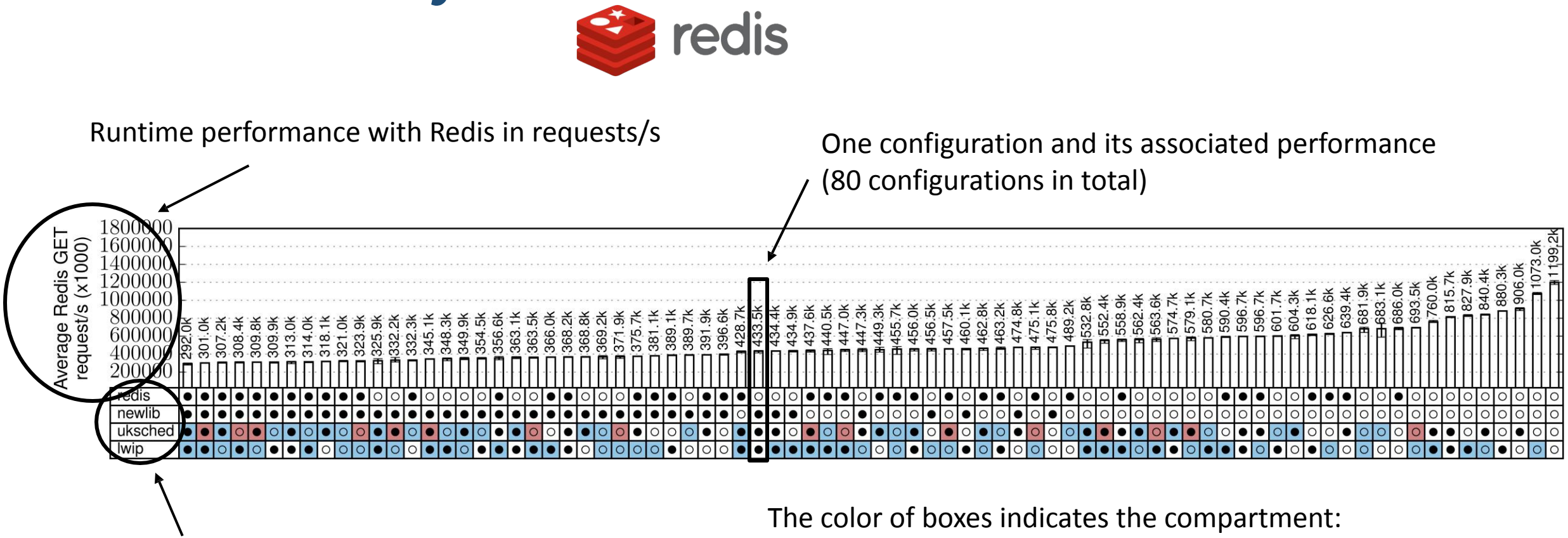

FlexOS libraries used in the Redis image
(only a subset for readability):

- Redis application
- C standard library (newlib)
- FlexOS scheduler (*uksched*)
- Network stack (lwip)

The color of boxes indicates the compartment:

Compartment 1 Compartment 2 Compartment 3

# Flexibility




FlexOS libraries used in the Redis image (only a subset for readability):

- Redis application
- C standard library (newlib)
- FlexOS scheduler (*uksched*)
- Network stack (lwip)

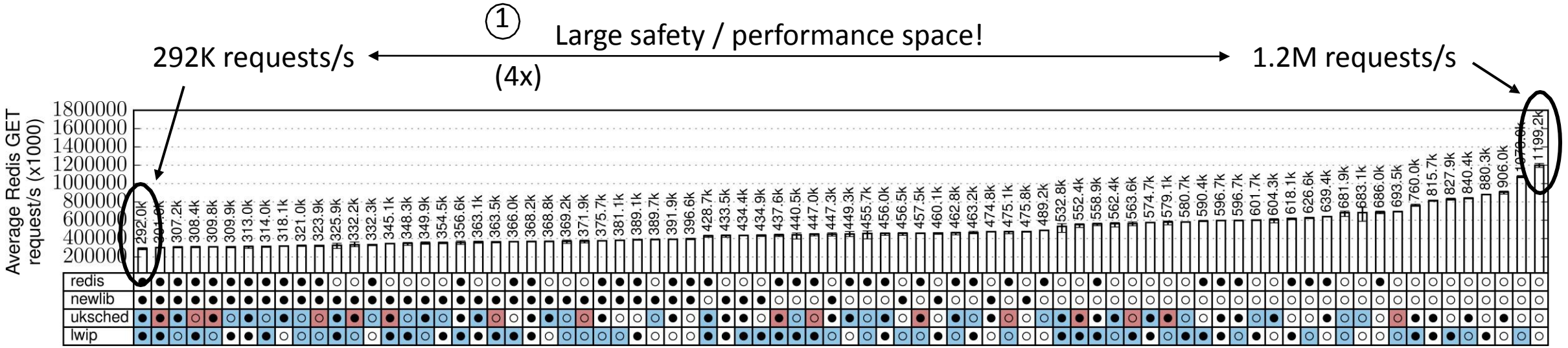
The color of boxes indicates the compartment:

Compartment 1 Compartment 2 Compartment 3

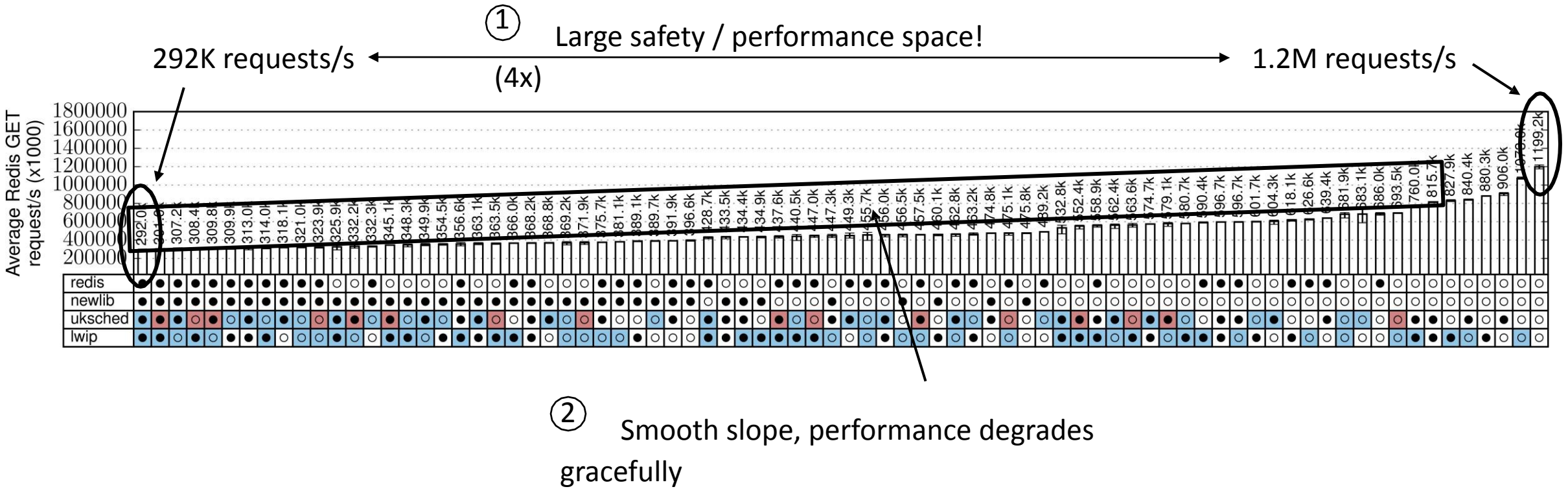
dot whether hardening (ASan, Safestack, etc.) is enabled:

Hardening on Hardening off

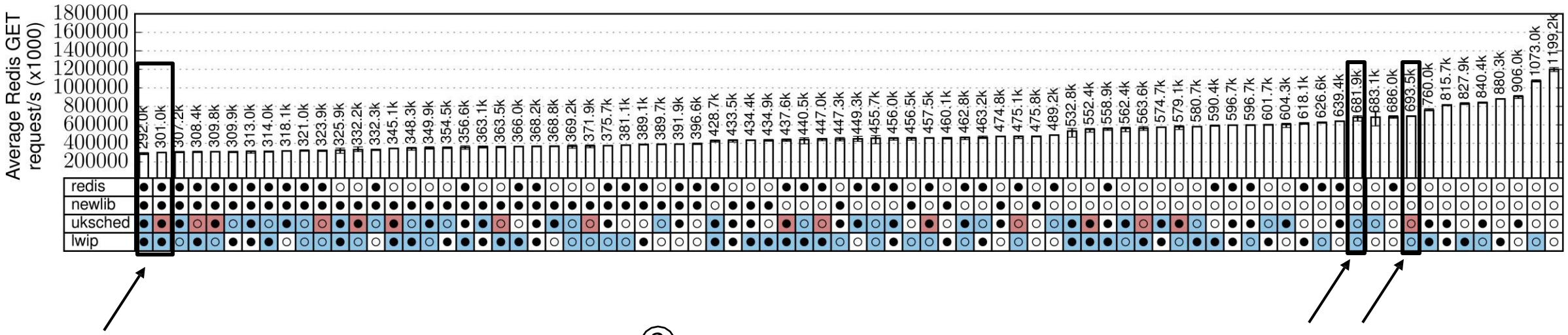
# Flexibility



# Flexibility



# Flexibility

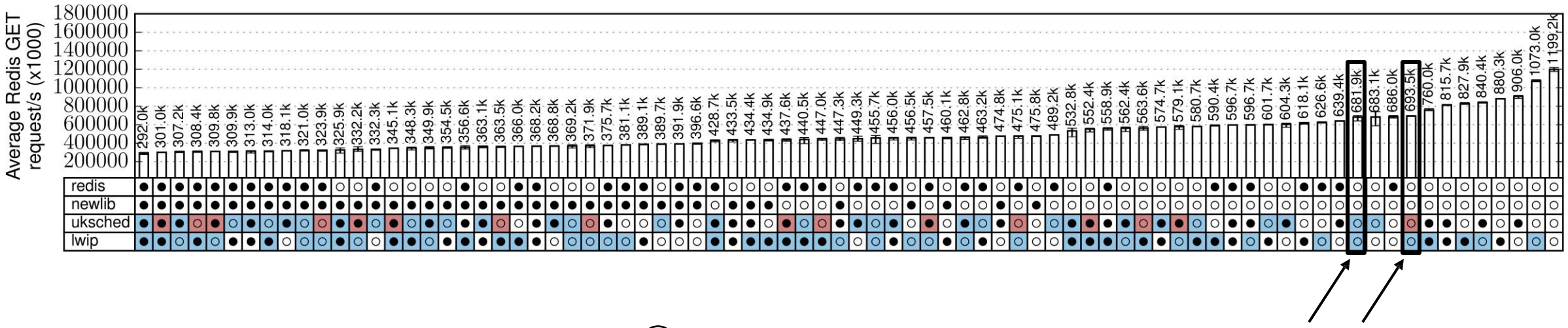



③ Similar performance, very different properties!

need to reason about communication patterns, fast paths

Hardening on | Hardening off | Compartment 1 | Compartment 2 | Compartment 3

# Flexibility

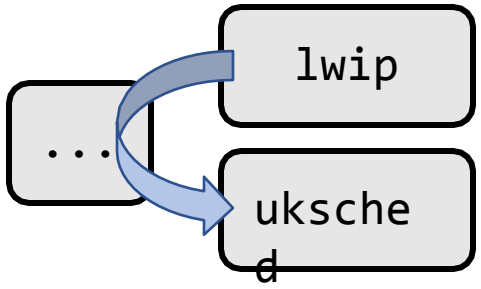


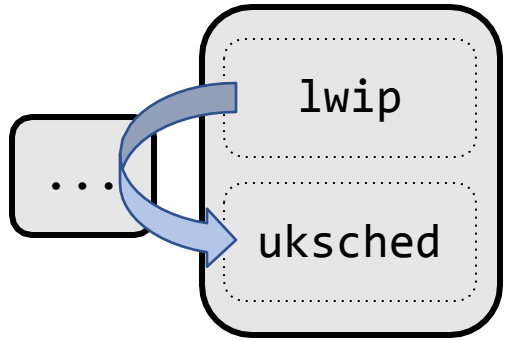

③ Similar performance, very different properties!

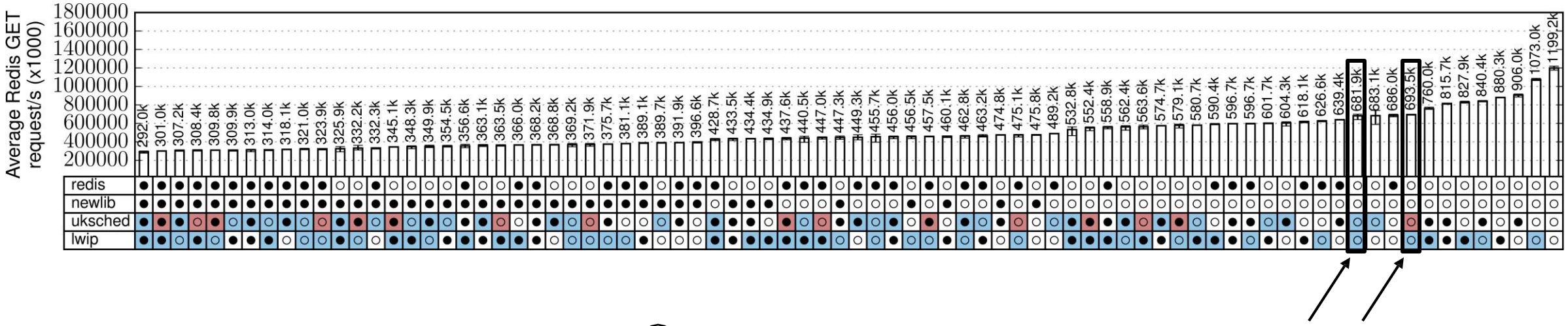
need to reason about communication patterns, fast paths

Hardening on | Hardening off | Compartment 1 | Compartment 2 | Compartment 3

# Flexibility



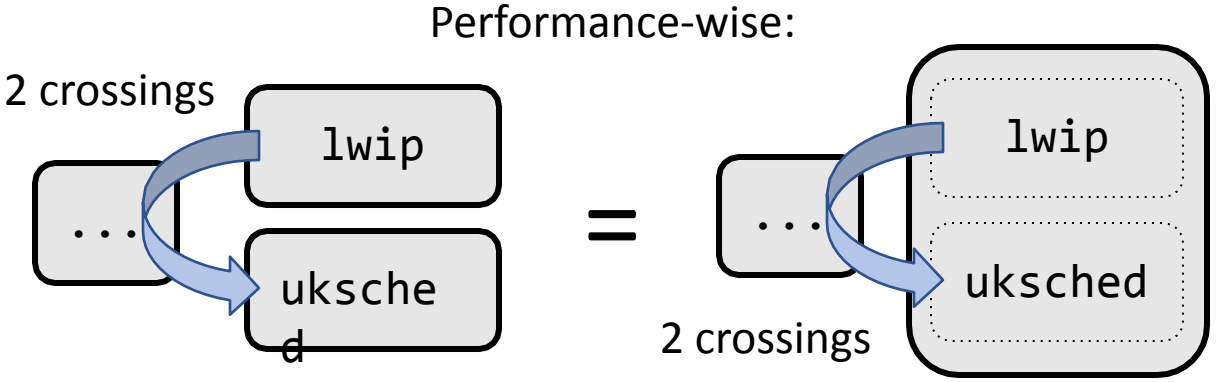



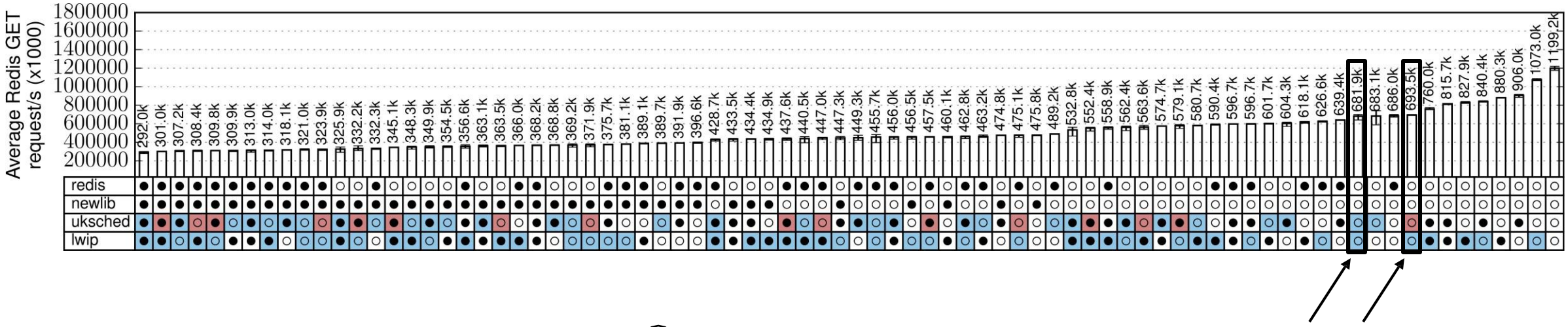

③ Similar performance, very different properties!

need to reason about communication patterns, fast paths

Hardening on | Hardening off | Compartment 1 | Compartment 2 | Compartment 3

# Flexibility



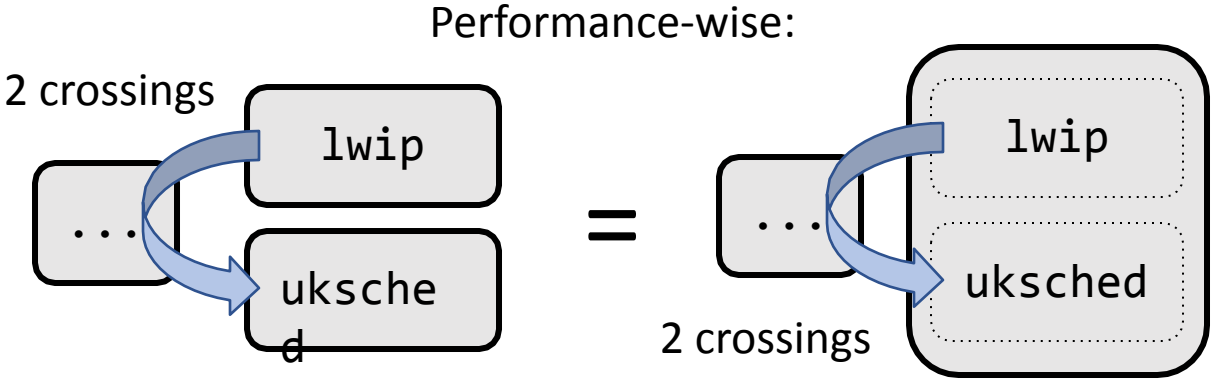



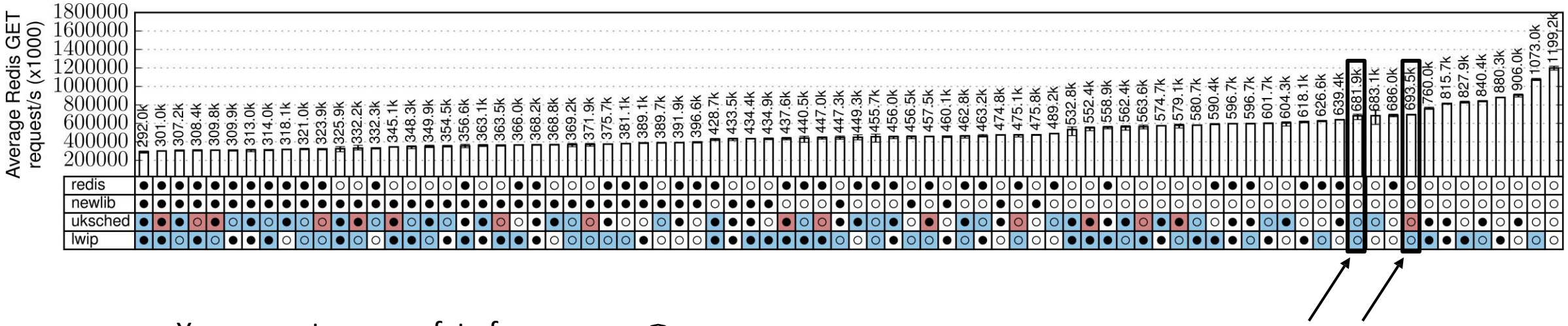

③ Similar performance, very different properties!

need to reason about communication patterns, fast paths

Hardening on | Hardening off | Compartment 1 | Compartment 2 | Compartment 3

# Flexibility




You can get some safety for free by exploring intelligently

③ Similar performance, very different properties!

need to reason about communication patterns, fast paths

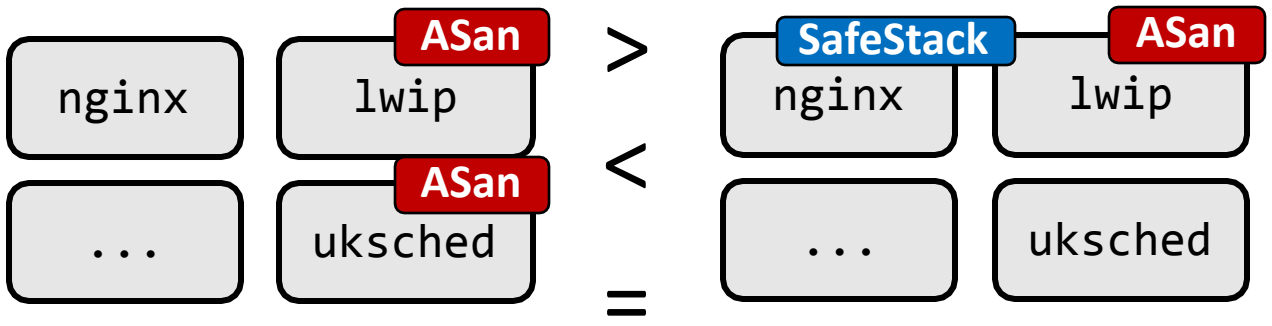
● Hardening on ○ Hardening off □ Compartment 1 ■ Compartment 2 ■ Compartment 3

# Exploring the design space

Now, we've a nice framework!

We can leverage FlexOS to get the most secure image for a given
performance budget!

Problem: some configurations are not comparable

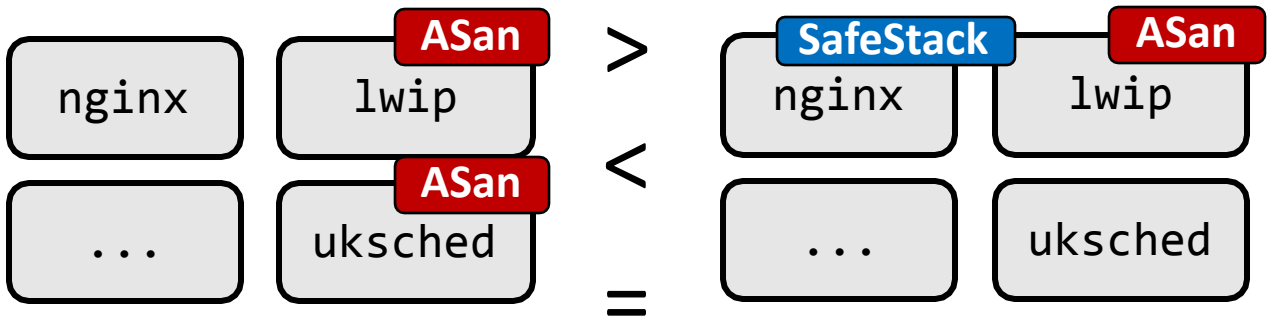



# Exploring the design space

Now, we've a nice framework!

We can leverage FlexOS to get the most secure image for a given
performance budget!

Problem: some configurations are not comparable




How can we reason about security/performance trade-offs?

# ASAN primer

- AddressSanitizer (ASan) detects memory errors by placing red zones around objects and checks those objects on trigger events
  - Out-of-bounds accesses to heap, stack and globals, use-after-free, use-after-return (configurable), use-after-scope (configurable) double-free, invalid free, memory leaks
- Mechanism: Uses “shadow memory” to track memory access validity, providing detailed memory reports when violation occur
  - Typical slowdown introduced by AddressSanitizer is 2x

# SafeStack primer

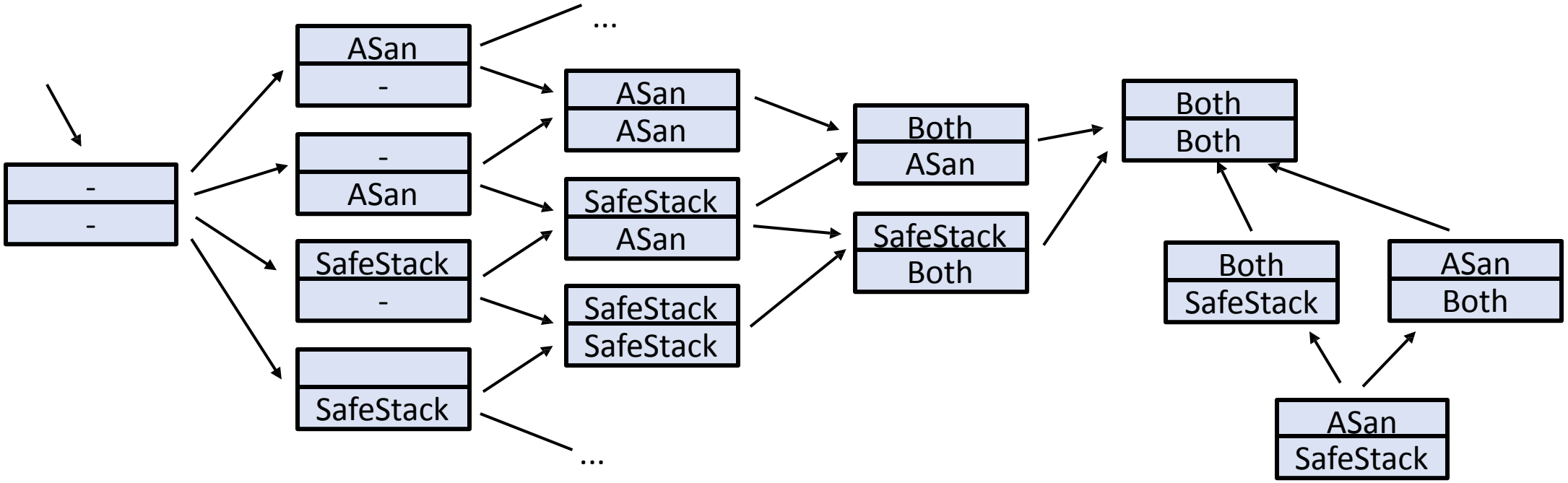
- Prevents stack-based buffer overflows by isolating critical data like the return address from the regular stack
- Mechanism: Allocate a separate “safe stack” region that stores sensitive data, preventing unauthorized access from malicious code attempting to overwrite the stack
- Less overhead than ASAN

# Exploring the design space

What we propose: consider configurations as a partially ordered set (poset)

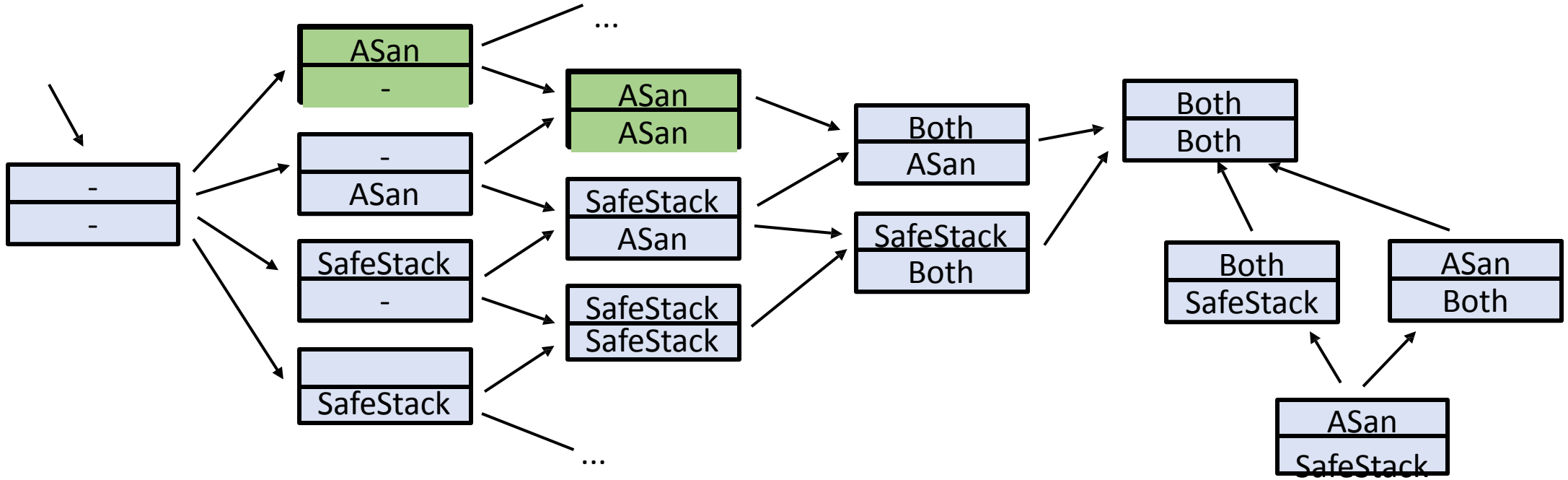
# Exploring the design space

What we propose: consider configurations as a partially ordered set (poset)



# Exploring the design space

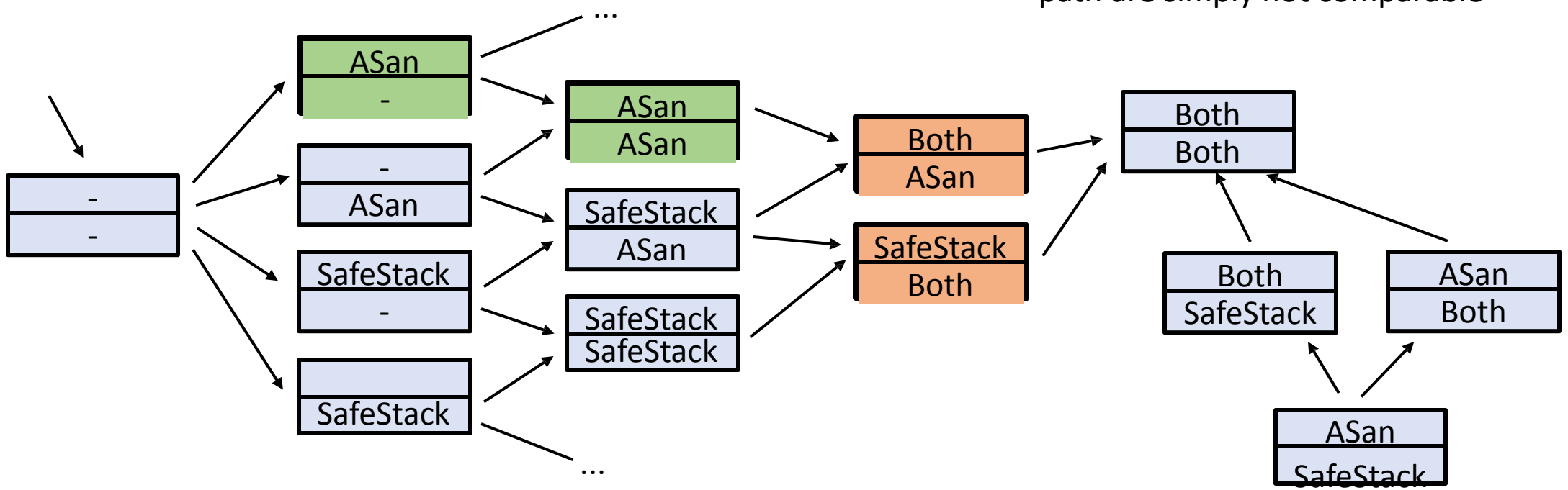
What we propose: consider configurations as a partially ordered set (poset)



# Exploring the design space

What we propose: consider configurations as a partially ordered set (poset)

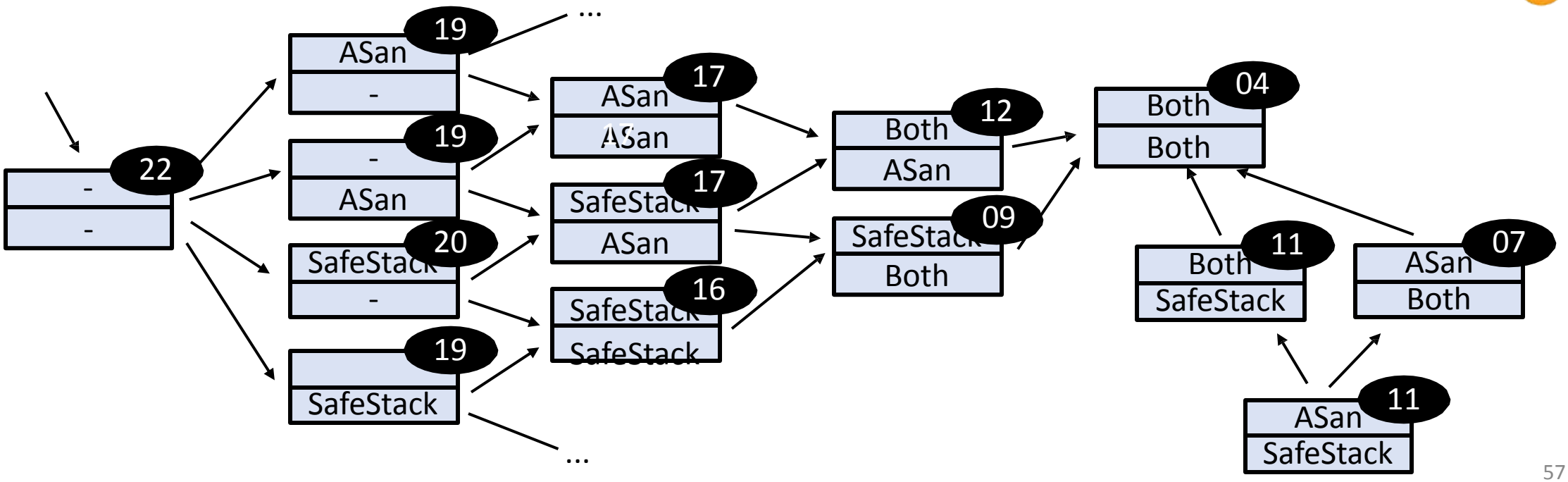
Two configurations that do not share a path are simply not comparable



# Exploring the design space

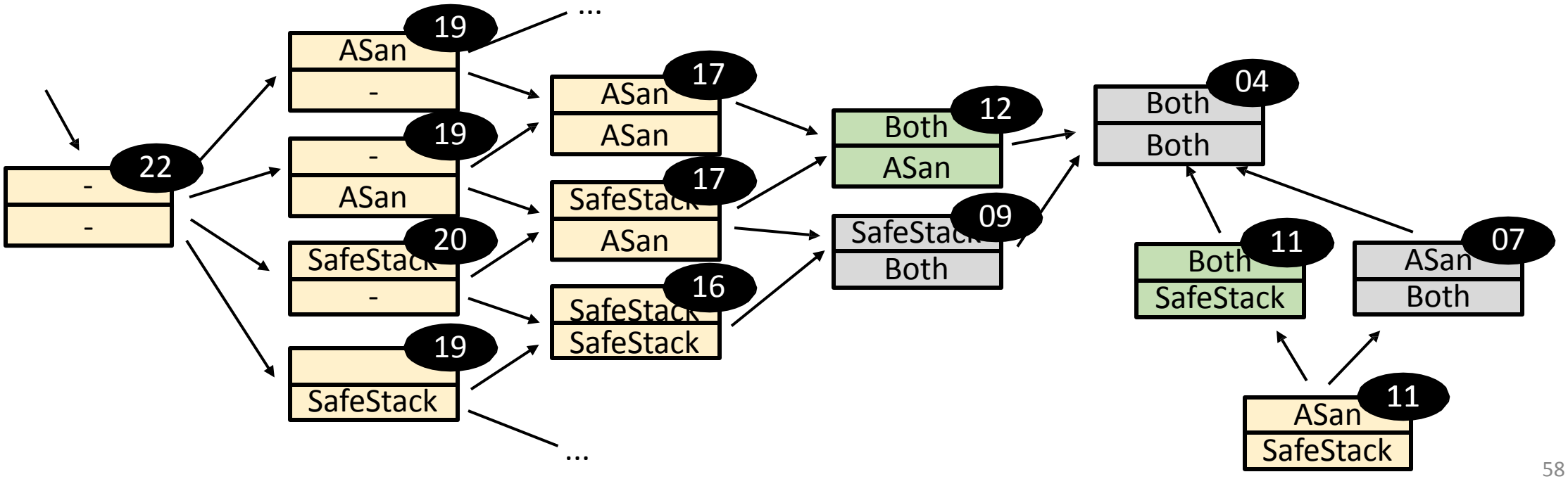
We can then label each node with performance characteristics (in practice no need to label everything)

Fictive numbers here 🙂



# Exploring the design space

Based on this ordering and labeling we can choose the last node of each path that satisfies the performance constraints



# Exploring the design space

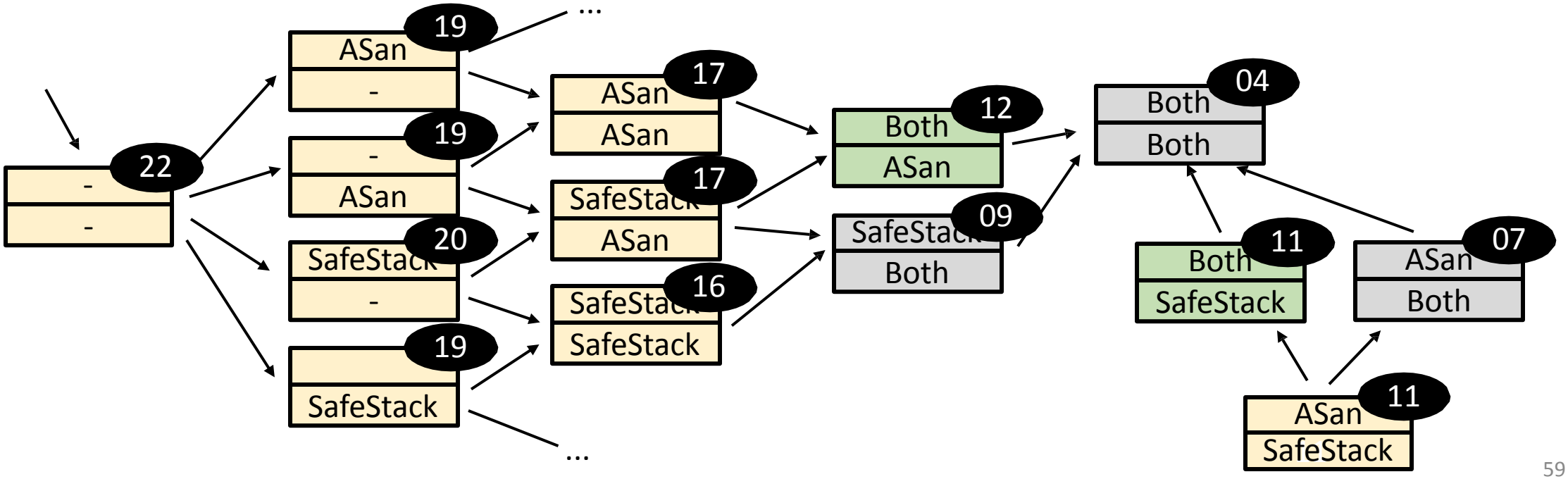
Let the user do the final choice

Based on this ordering and labeling we can choose the last node of each path that satisfies the performance constraints

| Both |
| --- |
| ASan |

| Both |
| --- |
| SafeStack |

Curated list of optimal configurations



# Exploring the design space

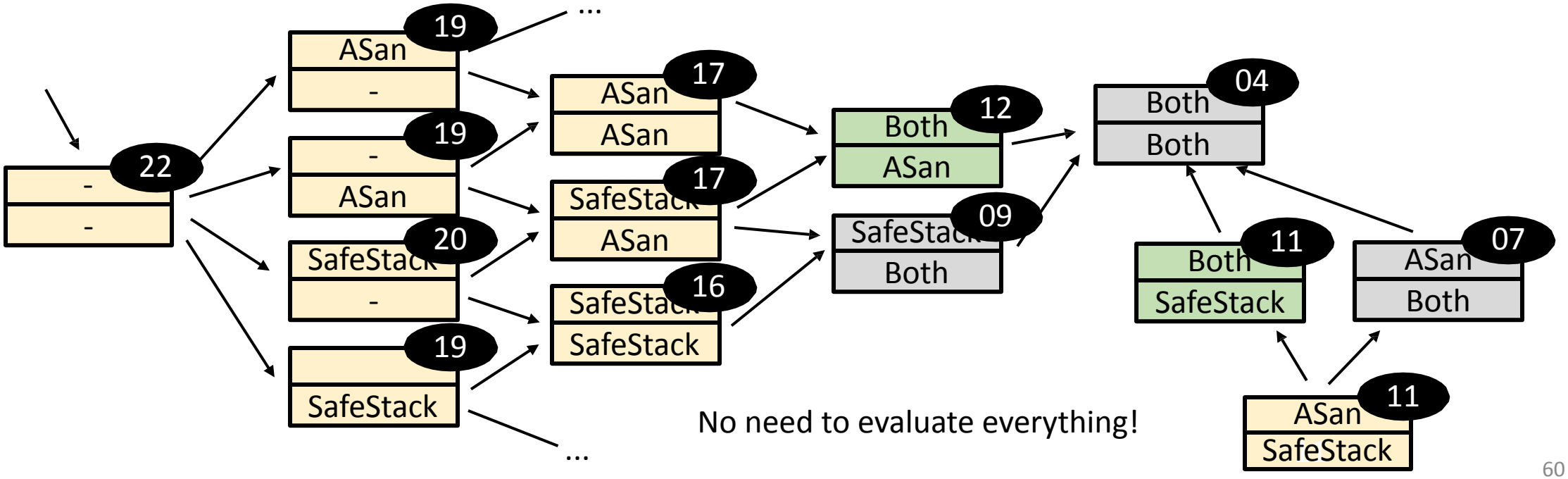
Let the user do the final choice

Based on this ordering and labeling we can choose the last node of each path that satisfies the performance constraints

| Both |
| --- |
| ASan |

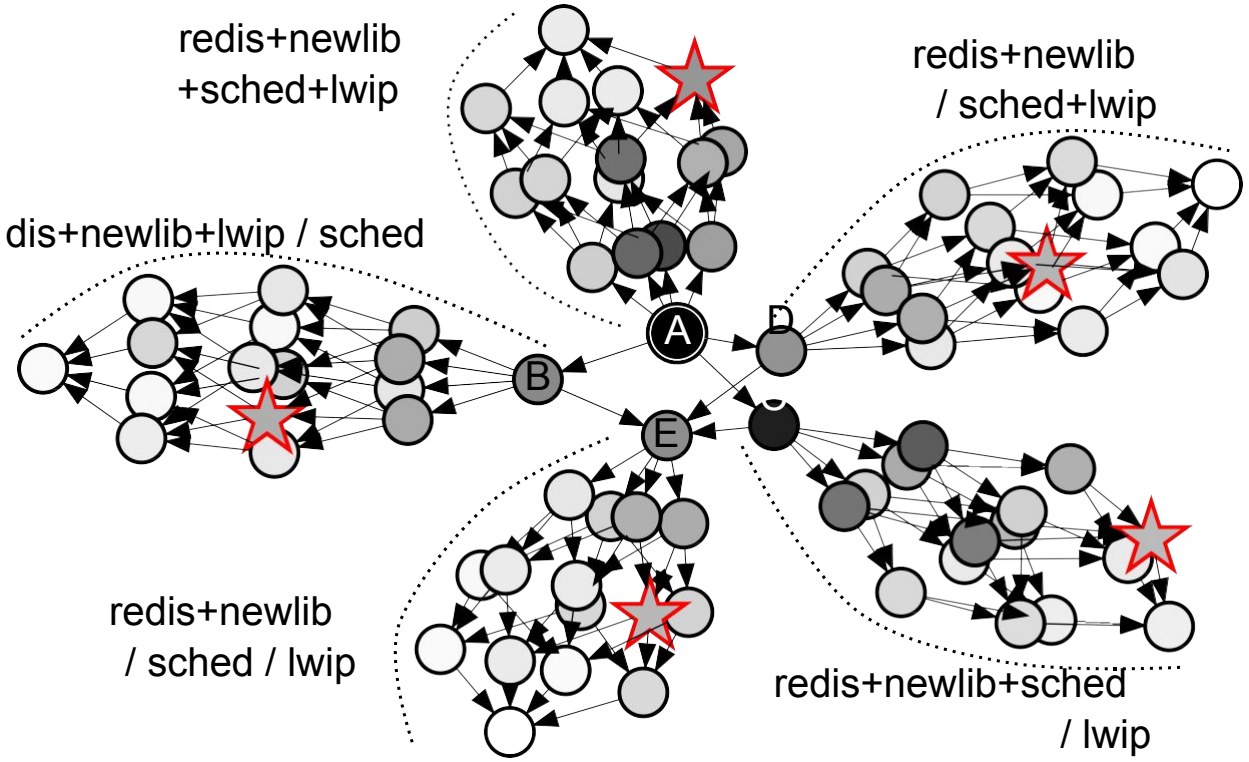
| Both |
| --- |
| SafeStack |

Curated list of optimal configurations




No need to evaluate everything!

# Applying POSet to Redis




**Reduction of 80 configurations to 5 candidates**

# Summary

- Security is now the first class citizen
- Different types of security mechanisms
- Providing performance and isolation is now critical
  - FLexOS provides a design point
  - Focuses on compartmentalization
  - Requires heavy compiler support