# Detecting Data Races Using Enhanced Memory Protection Keys


Abhirup Vijay Gunakar, Computer Science
Mentor: Dr. Adil Ahmad, ASTERISC Research Lab
School of Computing and Augmented Intelligence


## Introduction to Data Races

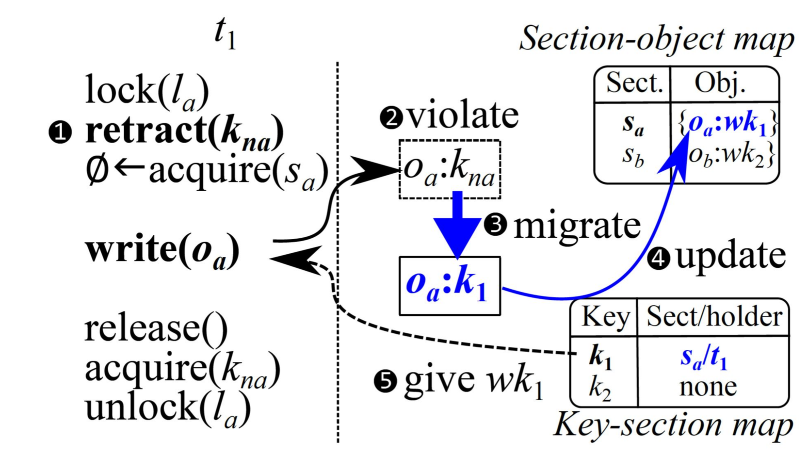
- A **data race** occurs when two threads access the same memory location concurrently, and at least one access is a write.
- **Inconsistent Lock Usage (ILU)** accounts for ~69% of real-world data races (based on TSan case studies).
- Tools like **ThreadSanitizer (TSan)** and **KCSAN** provide dynamic race detection, but suffer from high overhead or low accuracy due to sampling and instrumentation.
- Data races can lead to silent failures, system crashes, and security exploits in large-scale software.

## Key Enforced Race Detection Algorithm

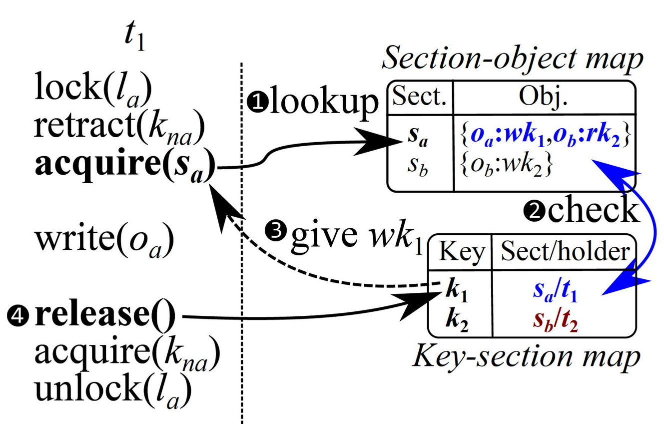
- **No Hardware Modifications:** KARD leverages Intel MPK to catch data races without compiler-based instrumentation.
- **Consolidated Page Allocation:** Small objects each get a unique virtual page, mapped into shared physical pages, enabling per-object isolation with minimal overhead.
- **Efficient Key Recycling:** KARD handles more than MPK's 16-key limit by carefully reusing keys across threads, avoiding excessive key sharing.

## Race Detection Pipeline

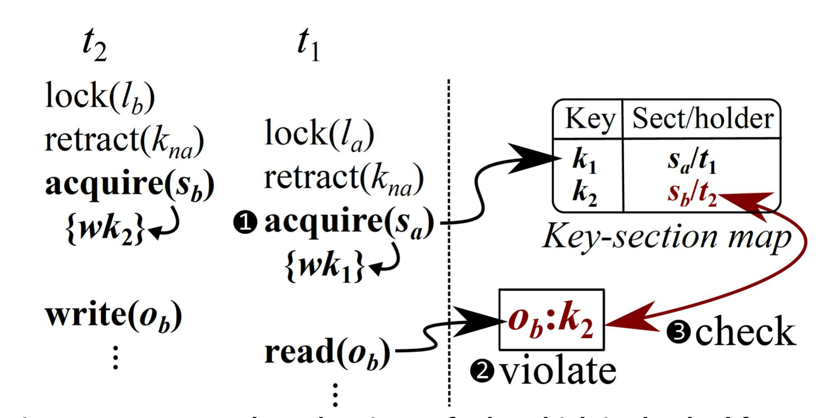
- Race detection is enforced through a three-stage pipeline: **a. object tracking**, **b. domain enforcement**, and **c. violation analysis**.
- **Shared objects** are assigned **protection keys** upon first access within a **critical section**.
- Access violations are inspected based on key ownership to determine the presence of true data races.




a. First access triggers a fault and object migration into a protected domain.




b. Protection keys are granted upon critical section entry based on tracked object.




c. Conflicting access across threads raises a fault, which is checked for race conditions.

## Objectives and Challenges

**Objectives**

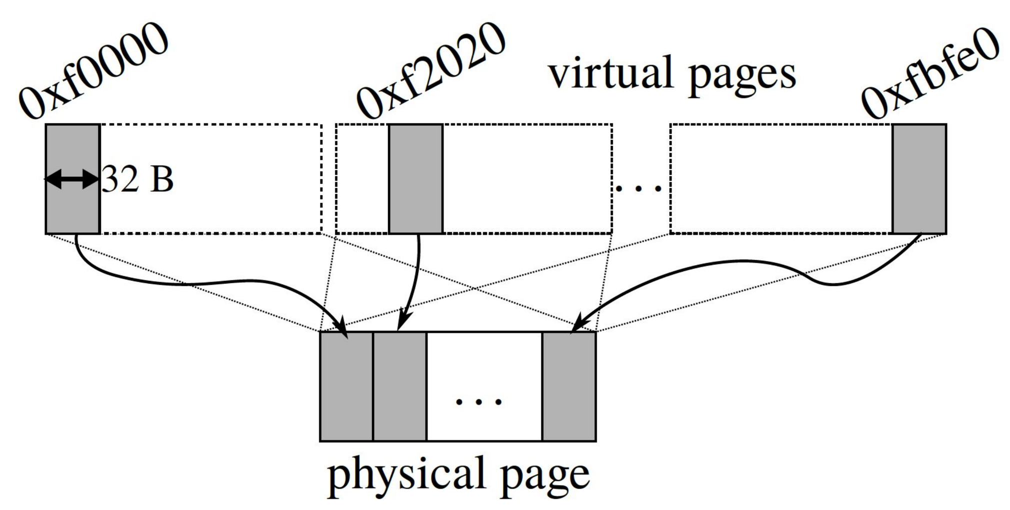
- Enable scalable and low-overhead detection of data races caused by inconsistent lock usage, without requiring instrumentation or hardware changes.

**Challenges**

- Intel MPK supports only 16 keys, limiting scalability for large applications.
- Protection operates at the page level, while most shared objects are much smaller.
- Maintaining accuracy with minimal performance and memory overhead remains non-trivial.

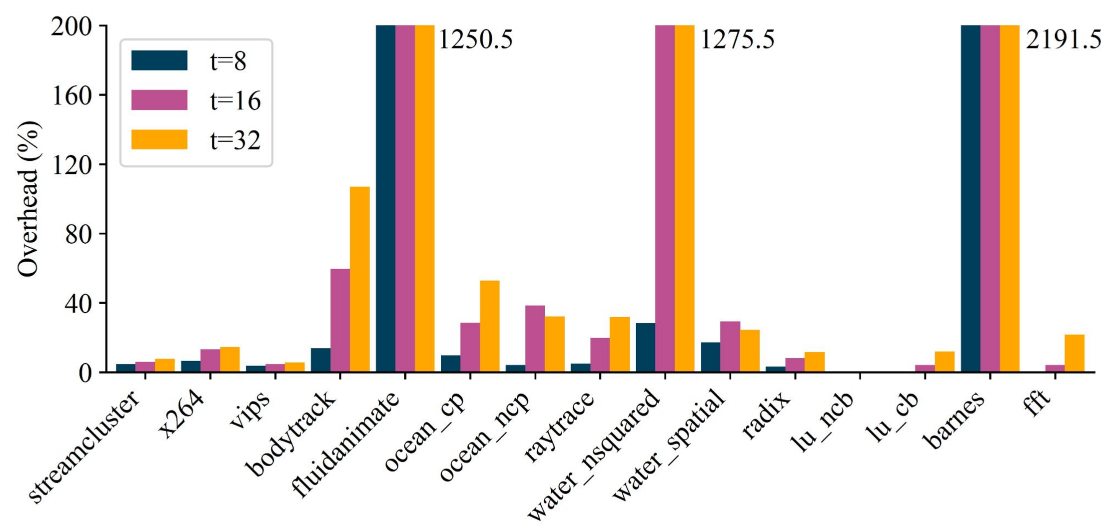
## Efficient Memory Isolation via Consolidated Page Allocation

- MPK enforces protection at the **4KB page level**, but many shared objects are much smaller.
- We allocate each object to a **unique virtual page**, allowing it to be independently protected.
- These virtual pages are then **mapped to a single physical page** using **`memfd_create()`** and **`mmap()`**, `by providing the correct offsets for each allocated object.`



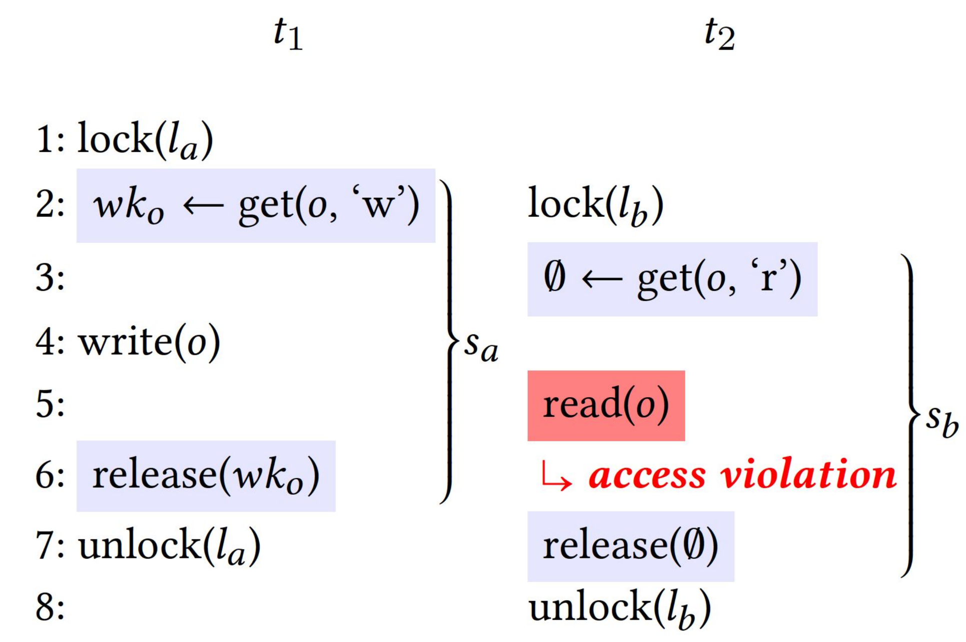

## Scalability and Performance Results

- Our system was evaluated using **PARSEC** and **SPLASH-2x** benchmark suites across **8, 16, and 32**-threads.
- **Maintains low overhead** for most benchmarks, confirming scalability.
- **Suitable for real-world environments** where performance is critical.
- **Delivers robust results** across diverse workloads, validating our protection strategy.

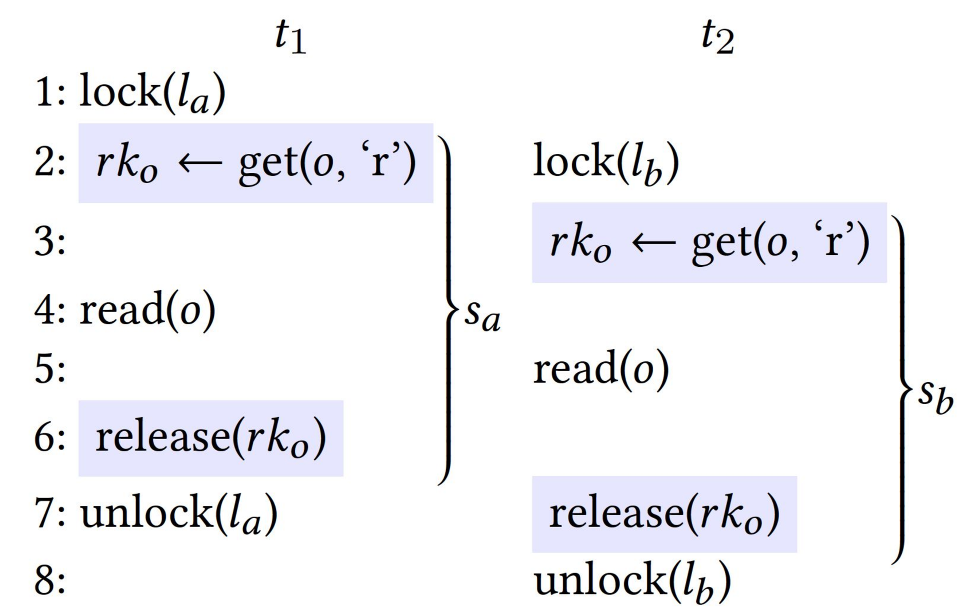



## System Architecture

- **Transparent Lock Wrappers:** Replaces standard lock/unlock with user-level calls that update PKRU, enforcing per-thread permissions.
- **PKRU-Based Memory Enforcement:** Threads grant/revoke read/write keys at critical sections, avoiding TLB flushes and instrumentation.
- **Consolidated Object Allocation:** A custom allocator gives each object a unique virtual page (or merges small ones with offsets) for MPK-based protection.
- **Fault-Driven Race Detection:** Unauthorized accesses trigger #GP faults; the handler logs potential races and discards false positives.




**(a) Exclusive write.** $t_2$ cannot obtain a read-only key $rk_o$ to read from $o$ while $t_1$ is holding a read-write key $wk_o$.




**(b) Shared read.** $t_2$ can obtain $rk_o$ to read from $o$ while $t_1$ is holding $rk_o$.

## Future Work and Discussion

- **Refine Extended Protection Keys (EPK)** by improving dynamic key mapping and reducing conflict rates under high concurrency.
- **Optimize key reuse and domain recycling** to further reduce runtime and memory overhead in long-running applications.
- **Integrate with kernel-space environments** and resource-constrained systems (e.g., embedded or real-time OS) to evaluate generality.
- **Explore hardware-assisted extensions** to scale beyond current MPK limitations and reduce performance overhead.