

Poster: Towards a Lightweight Key-Enforced Data Race Detector for Commodity Kernels

Rahul Priolkar
Arizona State University
rahul.priolkar@asu.edu

Chuqi Zhang
National University of Singapore
chuqiz@u.nus.edu

Adil Ahmad
Arizona State University
adil.ahmad@asu.edu

Abstract—Detecting data races in the kernel is critical for ensuring both the reliability and security of modern operating systems. However, existing kernel data race detectors, either involving heavy instrumentations or imprecise probabilistic-based sampling, face a persistent dilemma in balancing runtime efficiency with detection accuracy. In this work, we propose a lightweight solution leveraging hardware memory protection keys (Intel MPK), enforcing thread-shared object monitoring and access violations detection on-the-fly. To incorporate protection keys (at page granularity) with individual object access monitoring, we introduce the Consolidated Unique Page Allocator (CUPA) for the kernel. CUPA efficiently represents a single physical page into a diverse set of virtual pages associated with thread-shared objects, allowing individual protection key assignments for each object without significant memory overhead. Meanwhile, we leverage virtualization extensions to overcome the hardware limitation on the number of protection keys, supporting thousands of protection domains for objects simultaneously. Preliminary research has proven the functional correctness of CUPA, while results have shown a 90MB memory and $\leq 10\%$ performance overhead on real-world applications, respectively.

I. BACKGROUND

A data race occurs when multiple threads access the same memory object concurrently without appropriate synchronization mechanisms (e.g., mutual exclusion locks), and *at least* one of the accesses is a write. Data races produce unpredictable computational results, and can lead to reliability problems and security vulnerabilities [5]. While data races are a problem for all multi-threaded software, they are especially critical during operating system kernel execution, since they can lead to full system crashes and privilege escalation.

Given the criticality of kernel data races, prior systems have proposed techniques to dynamically detect them during testing. Such *dynamic data race detectors monitor memory object access behaviors across different threads and analyze their patterns*. Specifically, techniques like ThreadSanitizer (TSAN) [6] use compilers to instrument and monitor all read and write operations, but suffer from high performance overheads that significantly reduce testing efficiency. In contrast, the state of the art kernel data race detectors like KCSAN and DataCollider [2] use probabilistic sampling methods to monitor only a small set of shared objects at runtime. This ensures good performance, but reduces detection effectiveness.

Recent work [1] has made strides to address the performance and effectiveness trade-off in dynamic data race detectors for userspace programs using the idea of **key-enforced race detec-**

tion. The idea leverages efficient hardware memory protection keys (e.g., Intel MPK), a feature that allows restricting read and write permissions to different pages at a per-thread level. At a high-level, each shared object (e.g., heap or global) is bound using a protection key. Each thread must acquire the key before reading the shared object (e.g., when the thread enters a lock-region, also called a *critical section*). However, the system only grants the key to a thread, if and only if the key has not already been acquired by other threads. Accessing a shared object without its corresponding key results in a *protection key fault*, which is logged as a potential data race.

II. RESEARCH GOAL AND CHALLENGES

Given the promising use of hardware protection keys for enforcing object access pattern monitoring and prior wisdom in employing them for userspace data race detection, an intuitive question arises: “*Can a protection key-enforced mechanism be extended to support kernel-level data race detectors?*”

Our study on the kernel reveals the following challenges.

C1: Integrating key semantics into kernel heap allocators. Key-enforced race detection requires every shareable object to be *page-aligned* (since MPK can only set permissions at page-granularity [4]), and prior work designs a new userspace memory allocator for this task. Unfortunately, adopting a new memory allocator for commodity kernels is very challenging in practice, since allocators are a significant codebase and current methods are well-optimized for conserving physical memory space. Therefore, this requirement presents a significant deployment barrier for commodity kernels like Linux.

C2: Overcoming key limitations for full-scale race detection. With key-enforced race detection, in practice, each unique critical section requires at least one protection key. This is a challenge because modern hardware supports only up to 16 unique protection keys—unlike userspace programs with a limited number of critical sections [3], commodity kernels like Linux have thousands of unique critical sections. While there are techniques proposed to handle such limitations, they require transferring objects across keys and sharing keys, which leads to high overheads and false positive data races, respectively [1].

III. DESIGN

A. Consolidated Unique Page Allocator (CUPA) Frontend

Kernel memory allocators are composed of two components: (a) a *frontend* that allocates and frees virtual address ranges

to objects and (b) a *backend* that assigns physical pages to allocated objects while optimizing their utilization using several compression techniques. To enable existing kernel allocators to be protection key-aware without fully-redesigning them, we propose CUPA as a new kernel memory *frontend*.

At a high-level, CUPA leverages the following workflow. Initially, CUPA intercepts a request for each object allocation (e.g., using `kmalloc` in Linux), after the kernel backend has allocated a physical region for the object. Then, CUPA creates (and maintains) unique virtual pages for the requested object, mapped to its assigned physical region. This physical region can be shared with other objects; thus requiring no changes to the backend. Specifically, since MPK works at virtual page addresses [4], such sharing at the physical page-level does not harm protection key semantics. When the object is freed, CUPA returns its virtual page address to its internal list (explained next), while allowing the backend to reclaim the physical region.

One of the important optimizations required is to ensure the frontend can allocate and free virtual pages efficiently. To do so, CUPA follows the virtual page allocation strategy of other Linux kernel frontends, where it maintains *freelist structures* for different object sizes (e.g., at power of 2 intervals). Moreover, these lists are temporarily maintained at a *per-hardware thread* level to avoid continuous synchronization costs, while being synchronized at discrete intervals (e.g., when the memory allocation rate is low), or when lists are exhausted. Finally, we seamlessly integrated CUPA into the Linux SLUB allocator, and exposed a new flag `GFP_CUPA`. This required approximately 130 lines of code changes to the Linux SLUB implementation.

B. Virtualization-Extended Protection Keys

To tackle the hardware limitation of protection key numbers, we rely on a widely-available virtualization extension to unlock *thousands of virtualized protection keys* [3]. The solution involves (a) using multiple second-level address translation tables (particularly Intel EPT) to map different sets of protection key domains, and (b) using hardware-assisted fast translation table switching to change the activated protection keys.

When assigning a unique protection domain to a page, we extend its allocation by associating it with a distinct EPT entry alongside the legacy memory protection key ID. Pages within the same protection domain share both the associated EPT entry and the memory protection key ID. In contrast, pages not assigned to any protection domain are managed using the default protection key and native EPT entries. To quickly switch between protection domains, we employ the hardware-assisted EPT switching technique (namely `VMFUNC`), which enables changing active EPT entries without exiting to the hypervisor. Upon a protection domain switch, only the EPT entries associated with the target domain are activated, while pages belonging to other protection domains are marked as non-accessible. Therefore, the legacy protection key is virtualized by different sets of EPTs, allowing duplicated physical keys to be isolated into different logical domains.

IV. PRELIMINARY RESULTS

Our implementation of the design is based on the Linux kernel (with SLUB) and KVM hypervisor. Our preliminary evaluation shows a 90MB additional memory usage and $\leq 10\%$ performance overhead on real-world applications like OpenSSH. On system call intensive benchmarks like LMBench, we did notice higher overheads of upto $4\times$. With thorough analysis, we found that the majority of the performance overhead stems from the TLB flush associated with freed CUPA pages, before they are made available for future allocation requests. In the future, we expect an implemented batched TLB flushing on CUPA pages would significantly reduce such overheads.

V. DISCUSSION AND FUTURE WORK

Currently, our system allocates unique page objects and has the ability to set permissions for each object, but does not detect races. Therefore, the first step is to integrate our protection key mechanisms with data race detection primitives. Specifically, like prior work [1], the idea is to initially rely on explicit lock-based synchronization primitives (critical sections) to identify when shared objects are supposed to be accessed and proactively assign keys to detect races. To this end, we have identified all the locking primitives offered by the kernel, and instrumented two of the most frequently used ones i.e. `mutex_lock()` and `spin_lock()`. However, we also intend to improve upon lock-oriented protection, because this approach fails to *account for data races in non-critical section shared objects* ($\approx 30\%$ of data races according to prior studies). At a high-level, we will achieve this by treating the code region between an `unlock` and a subsequent `lock` as another critical section rather than ignoring it; thereby detecting data races in all code regions. Finally, since the system is designed for testing, we will next integrate it into the testing pipeline of concurrency-aware fuzz testing frameworks.

VI. CONCLUSION

Achieving the best of both worlds between performance and detection effectiveness is a challenge for current kernel dynamic data race detectors. This work provides a first step to addressing this challenge using efficient hardware protection keys, solving challenges in their use within the kernel and providing a design roadmap towards a full data race detector.

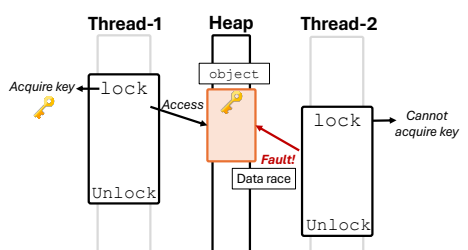
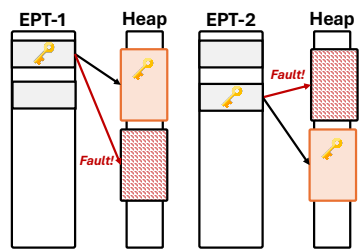
REFERENCES

- [1] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: Lightweight Data Race Detection with Per-thread Memory Protection. In *ACM ASPLOS*, 2021.
- [2] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX OSDI*, Vancouver, Canada, 2010.
- [3] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and efficient memory protection keys. In *USENIX ATC*, 2022.
- [4] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: *System Programming Guide*, 2016.
- [5] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*.
- [6] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

Towards a Lightweight Key-Enforced Data Race Detector for Commodity Kernels

Rahul Priolkar¹ Chuqi Zhang² Adil Ahmad¹

¹Arizona State University ²National University of Singapore

Introduction to Data Races	Goal and Challenges	Preliminary Results
<ul style="list-style-type: none"> Occurs when multiple threads access the same memory region concurrently, without a lock, and at least one access is a “write” Can lead to system crashes and even privilege escalation if races happen during kernel execution 	<ul style="list-style-type: none"> Goal: Bring key-enforced race detection to commodity kernels Challenges: <ol style="list-style-type: none"> Integrating key semantics into kernel heap allocators to page align all heap objects Overcoming key limitations for full-scale race detection 	<ul style="list-style-type: none"> Implemented design elements for Linux kernels and using the KVM hypervisor Low overheads $\leq 10\%$ on regular programs; 4x overheads on LMBench (<i>amortized during execution</i>) 90MB of more memory required
<h3>Dynamic Kernel Data Race Detectors</h3> <ul style="list-style-type: none"> Cannot give the best of both worlds between performance and detection effectiveness Techniques like ThreadSanitizer [1] incur high overheads through compiler instrumentation KCSAN and DataCollider [2] sample events leading to lower detection effectiveness 	<h3>Design: Page-Aligned Memory Frontend</h3> <ul style="list-style-type: none"> Allocates kernel heap objects on unique virtual pages Maintains <i>freelists</i> for different object sizes, for efficient virtual page assignment Uses kernel memory allocator backend for physical page allocation 	<h3>Future Work and Discussion</h3> <ul style="list-style-type: none"> Extend with data race detection primitives based on Kard [3] Improve lock-oriented protection by incorporating unlock-lock semantics Integrate into a concurrency testing system to show performance advantage
<h3>Key-Enforced Race Detection Algorithm</h3> <ul style="list-style-type: none"> Leverages efficient hardware protection keys to detect simultaneous memory access [3]  <p>The diagram shows two threads, Thread-1 and Thread-2, accessing a shared heap object. Thread-1 acquires a key (represented by a yellow key icon) and locks the object. Thread-2 attempts to access the object but cannot acquire the key, leading to a 'Data race' and a 'Fault!'.</p>	<h3>Design: Virtualized Kernel Protection Keys</h3>  <p>The diagram shows two virtual machines, EPT-1 and EPT-2, each with its own heap. EPT-1 has a key and accesses its heap object. EPT-2 attempts to access its heap object but encounters a 'Fault!' because it does not have the key.</p>	<h3>References</h3> <p>[1] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA), 2009.</p> <p>[2] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010</p> <p>[3] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: Lightweight Data Race Detection with Per-thread Memory Protection. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021</p>