KARD: Lightweight Data Race Detection with Per-Thread Memory Protection
Extended Abstract

Adil Ahmad* Sangho Lee† Pedro Fonseca* Byoungyoung Lee‡
*Purdue University †Microsoft Research ‡Seoul National University

1. Motivation

Data races are frequently responsible for hard-to-diagnose concurrency bugs. A promising approach to detect data races is to automatically monitor and analyze a program's concurrent behavior during execution. In particular, dynamic data race detectors monitor each memory read and write access as well as synchronization operations performed by each thread to detect potentially conflicting and racing accesses. Dynamic data race detectors have been successful at detecting many data races in widely-used programs (e.g., Firefox and Chromium web browsers [21]) with low or no false positives.

2. Limitations of the State of the Art

The state-of-the-art dynamic data race detector, ThreadSanitizer (TSan) [19], incurs a prohibitive performance overhead (7×) because it employs expensive compiler instrumentation to monitor every memory access. Despite being one of the most commonly used data race detectors, TSan is not efficient enough for all development and testing settings and is generally hardly employed by developers [12].

Although several schemes mitigate the performance overhead of data race detection, they all have critical scope, deployability, or automation limitations. First, sampling schemes [5, 9, 20, 22] select a subset of instructions and objects to monitor, resulting in probabilistic guarantees. In practice, when sampling schemes are configured for low overhead, they only detect a tiny fraction of data races. Second, other schemes [14, 17, 23, 24] require substantial system software or hardware changes, which hinders deployment. Lastly, some schemes require significant, error-prone developer effort, e.g., almost 35 hours of manual annotation for a single application [23], to selectively monitor memory objects.

3. Key Insights

Intel Memory Protection Keys (MPK), a new hardware feature available in commodity Xeon and Icelake CPUs [1, 15], enables per-thread protection of a program's memory. Importantly, using MPK, programs can change permissions for a memory region with high efficiency (i.e., 23 cycles [18]).

Our key insight is that we can use per-thread memory protection to exclusively assign shared memory regions, during synchronization operations (e.g., `lock()`), to a particular thread and detect potential data races. Once a memory region is exclusively assigned to a thread, any access from other threads will result in an access violation, signaling a potential data race bug.

This paper proposes KARD, a low-overhead dynamic data race detector that leverages per-thread memory protection [3, 6, 10]. KARD detects data races caused by inconsistent lock usage (§4.1), which occurs when a program concurrently accesses the same shared object using different locks or only some of the concurrent accesses are synchronized using a common lock. We analyze real-world data races detected by TSan and eventually fixed by developers, and confirm that inconsistent lock usage constitutes 69% of them. In contrast to existing schemes, KARD does not rely on expensive compiler instrumentation for every memory operation, random sampling, changes of the system's hardware or software, or program annotations. We show that KARD can detect all data races due to inconsistent lock usage with low false positive, for our evaluated real-world applications, while incurring a geometric mean performance overhead (using 4 threads) of only 5.23%.

4. Main Artifacts

This paper contributes three main artifacts: (a) a key-enforced data race detection algorithm that detects data races due to inconsistent lock usage, (b) a dynamic data race detector, KARD, that implements key-enforced access and addresses the limitations of MPK, and (c) the implementation of KARD, an LLVM instrumentation component and a runtime library, and its evaluation on micro-benchmarks and real-world applications.

Key-enforced data race detection algorithm. This paper introduces a new algorithm, based on *key-enforced access*, to detect data races caused by inconsistent lock usage (§5). With key-enforced access, a thread that enters a lock-protected region (i.e., critical section) must acquire either (a) an *exclusive* key to *write* to an object, provided no other thread has any key to the object, or (b) a *shared* key to *read* from an object, provided no other thread has an exclusive key to the object. The thread releases the acquired key(s) when it exits the critical section. Thus, while a thread is holding the key to an object, other threads' conflicting accesses to the object are flagged as potential data races.

KARD: Dynamic data race detector. KARD employs key-enforced access and overcomes three main challenges to detect data races given the limitations of commodity per-thread memory protection. First, KARD must identify shared objects that are accessed within critical sections to protect such objects during execution. Second, per-thread memory protection operates at the granularity of memory pages (typically 4 KiB), but native memory allocation assigns multiple objects to the same

page. Third, current per-thread memory protection hardware supports only 16 protection keys, which is too small for highly multi-threaded programs.

To overcome these limitations, KARD implements three techniques: automated shared object identification (§6.3), consolidated unique page allocation (§6.3), and effective key assignment (§6.4).

Automated shared object identification. KARD implements a shared object tracking scheme to accurately, automatically, and progressively identify objects accessed in critical sections. In particular, KARD protects each newly created heap and global object with a key that a thread, executing a critical section, initially cannot acquire. Hence, the first access to a protected object, in a critical section, causes an access violation that is caught and handled by KARD.

Consolidated unique page allocation. KARD's custom memory allocator assigns unique virtual pages to each heap and global memory allocation, enabling individual protection of each object using MPK. Moreover, to minimize physical memory usage, the allocator maps several virtual pages storing small-sized objects into the same physical page by shifting their page-internal offsets, similarly to *minipage* or *page-aliasing* techniques.

Effective key assignment. KARD carefully uses a limited number of protection keys available in existing per-thread protection mechanisms (e.g., 16 keys in MPK). Specifically, KARD reuses a protection key that was acquired within a critical section or is already held by a thread. When key reuse is not possible, KARD assigns an unused protection key, recycles a protection key that is not currently held by any thread, or shares a protection key between threads, while minimizing potential issues.

KARD employs various methods to automatically analyze access violations and prune out those caused by non-racy events during execution. For instance, it uses a new *protection interleaving* scheme to obtain fine-grained access information from conflicting threads to test if the violation is due to a data race (§6.5).

Implementation and evaluation. We implement KARD (§7) using (a) an LLVM compiler [11] pass to trap heap allocations and synchronization calls and (b) a C++-based runtime library that creates unique paged heap objects and assigns protections to executing threads. KARD's implementation consists of 2850 lines of code written in C/C++. Furthermore, we evaluate KARD (§8) using the well-known PARSEC and SPLASH-2x benchmarks [4], as well as four real-world applications: NGINX [16], memcached [7], pigz [2], and Aget [8].

5. Key Results and Contribution

Our evaluation shows that KARD incurs a very low performance overhead. The geometric mean performance overhead of KARD, with 4 threads, is 7.42% for PARSEC and SPLASH-2x benchmarks and 5.23% for the four real-world applications. In comparison, the overhead of TSan for these workloads is 690.96% and 189.48%, respectively. We used a 4-thread configuration because it is a standard testing configuration that can reveal most data races [13]. However, even with 32-thread configuration, which utilizes all hardware threads on our machine, 10 out of the 15 evaluated benchmarks showed less than 30% overhead, illustrating the scalability aspect of KARD.

Furthermore, we provide an extensive analysis of potential false positives and negatives of KARD, as well as a detailed effectiveness evaluation. In particular, KARD detected all five data races caused by inconsistent lock usage in our evaluated real-world applications and reported only one false positive. Hence, KARD is a low-overhead yet effective data race detector, suitable in many development and testing settings.

The key contributions of this paper are:

- **Key-enforced data race detection algorithm.** We propose a new algorithm that requires threads to acquire keys before accessing objects in critical sections and captures data races due to conflicting access on such objects.
- **KARD: Dynamic data race detector.** KARD employs our proposed algorithm, using Intel MPK, to detect data races due to inconsistent lock usage. KARD does not require developer aid, system changes, or sampling.
- **Implementation and evaluation.** We implement KARD, demonstrate that it is effective at detecting data races, and show that it is lightweight—geometric mean of performance overhead is 7.42% for benchmarks and 5.23% for real-world applications, under common testing scenarios.

6. Why ASPLOS?

KARD uses a recent architectural feature (i.e., per-thread memory protection) to efficiently tackle a hard and long-standing programming language problem (i.e., data races in thread-unsafe execution). Further, KARD overcomes the limitations of current memory protection hardware by implementing software techniques such as minipage and effective key assignment. Therefore, we believe that KARD is well-suited to the inter-disciplinary focus of ASPLOS.

7. Citation for Most Influential Paper Award

"This paper describes a major step towards lightweight detection of data race bugs, a particularly challenging class of bugs that plagued software in the early days of the multi-core era. In contrast with prior techniques, KARD did not require expensive compiler instrumentation for every memory access, hardware changes, or developer effort. Instead, KARD was the first practical system to use per-thread memory protection hardware to automatically detect data races caused by inconsistent lock usage. Unlike prior techniques, KARD caused minimal overhead on real-world applications. The lightweight approach of KARD for a longstanding problem propelled the adoption of data race detectors in testing settings and significantly contributed to improve the reliability of programs."

References

[1] https://openbenchmarking.org/system/1909082-HV-ICELAKETE36/TW20190905/cpuinfo.

[2] Mark Adler. pigz - Parallel gzip. https://zlib.net/pigz/.

[3] ARM Developer. ARM922T Technical Reference Manual: Domain access control. https://developer.arm.com/docs/ddi0184/latest/memory-management-unit/domain-access-control.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[5] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, ON, June 2010.

[6] Jonathan Corbet. Memory protection keys, 2015. https://lwn.net/Articles/643797/.

[7] Dormando. memcached - a distributed memory object caching system. https://memcached.org.

[8] EnderUNIX Software Development Team. EnderUNIX Aget: Multithreaded HTTP Download Accelerator. http://www.enderunix.org/aget/.

[9] 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)*, Vancouver, Canada, October 2010.

[10] IBM. Storage protect keys. https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/storage_protect_keys.htm.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, 2004.

[12] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[13] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, March 2008.

[14] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proceedings of the 37th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Jose, California, USA, June 2010.

[15] David Mulnix. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[16] NGINX Inc. NGINX High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com.

[17] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

[18] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, June 2019.

[19] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[20] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications. In *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2007.

[21] Dmitry Vyukov. Threadsanitizerfoundbugs. https://github.com/google/sanitizers/wiki/ThreadSanitizerFoundBugs.

[22] Tong Zhang, Dongyoon Lee, and Changhee Jung. ProRace: Practical Data Race Detection for Production Use. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[23] Diyu Zhou and Yuval Tamir. PUSh: Data Race Detection Based on Hardware-Supported Prevention of Unintended Sharing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Columbus, OH, October 2019.

[24] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *Proceedings of the 15th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Raleigh, NC, USA, February 2009.