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

Adil Ahmad
Purdue University, USA

Sangho Lee
Microsoft Research, USA

Pedro Fonseca
Purdue University, USA

Byoungyoung Lee
Seoul National University, South Korea

ABSTRACT

Finding data race bugs in multi-threaded programs has proven challenging. A promising direction is to use dynamic detectors that monitor the program's execution for data races. However, despite extensive work on dynamic data race detection, most proposed systems for commodity hardware incur prohibitive overheads due to expensive compiler instrumentation of memory accesses; hence, they are not efficient enough to be used in all development and testing settings.

KARD is a lightweight system that dynamically detects data races caused by inconsistent lock usage—when a program concurrently accesses the same memory object using different locks or only some of the concurrent accesses are synchronized using a common lock. Unlike existing detectors, KARD does not monitor memory accesses using expensive compiler instrumentation. Instead, KARD leverages commodity per-thread memory protection, Intel Memory Protection Keys (MPK). Using MPK, KARD ensures that a shared object is only accessible to a single thread in its critical section, and captures all violating accesses from other concurrent threads. KARD overcomes various limitations of MPK by introducing key-enforced race detection, employing consolidated unique page allocation, carefully managing protection keys, and automatically pruning out non-racy or redundant violations. Our evaluation shows that KARD detects all data races caused by inconsistent lock usage and has a low geometric mean execution time overhead: 7.0% on PARSEC and SPLASH-2x benchmarks and 5.3% on a set of real-world applications (NGINX, memcached, pigz, and Aget).

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Multithreading**; **Virtual memory**; **Concurrency control**.

KEYWORDS

concurrency, data race, lock, memory protection keys

Adil Ahmad conducted part of this research as an intern at Microsoft Research.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446727

ACM Reference Format:
Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. 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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446727

1 INTRODUCTION

Dynamic data race detectors find real-world data races with low false positives [58], but generally incur prohibitive performance overheads. For example, the state-of-the-art dynamic data race detector, Google's ThreadSanitizer (TSan) [52], employs compiler-based memory access instrumentation that slows down program execution by almost 7×. Due to its prohibitive overhead, TSan is not applicable to all development or testing settings [34].

Unfortunately, existing approaches to improve the performance of data race detection have seen limited adoption due to their deployment and effectiveness limitations. Some schemes require system software modifications [42, 64], hardware modifications [38, 65] or extensive developer effort [21, 64], which hinder deployability. Others employ sampling [13, 20, 53, 63] that results in probabilistic guarantees and needs to be calibrated with a low sampling rate, hence low effectiveness, to ensure adequate performance.

This paper proposes KARD, a low-overhead yet practical dynamic data race detector that does not require hardware or system software changes, developer effort, and sampling. KARD detects data races due to *inconsistent lock usage* [28] that occurs when a program concurrently accesses the same memory object using different locks or only some of the concurrent accesses are synchronized using a common lock. Our analysis of fixed real-world data races detected by TSan shows that inconsistent lock usage constitutes approximately 69% of all reported data races (§3.1). Furthermore, under the standard testing scenario of 4 threads [36], KARD incurs a performance overhead of only 5.3% across a range of real-world applications. Hence, KARD is an effective data race detection tool that is applicable in many testing and development settings.

We introduce a novel data race detection algorithm based on the concept of *key-enforced access* (§4). KARD uses this algorithm to detect data races caused by inconsistent lock usage. In particular, key-enforced access protects each shared object accessed in a lock-protected code region (i.e., a *critical section*) using a *key*. While a thread holds a shared object's key within a critical section, all conflicting concurrent access (with a different lock or without a lock) to the object is flagged as a data race.

KARD implements the algorithm efficiently by leveraging Intel's Memory Protection Keys (MPK) [14], a hardware feature available on commodity CPUs [1, 39]. MPK allows software to change each thread's page access permissions by modifying a thread-local register (PKRU) with a non-privileged instruction (WRPKRU). Importantly, updating PKRU does not flush the Translation Lookaside Buffer (TLB); hence, updating memory protection using MPK is very efficient, as noted by other studies [26, 43, 57].

Nevertheless, KARD must overcome three important challenges including those presented by the hardware limitations of MPK. First, KARD must accurately track each shared object accessed in a critical section. Second, KARD must protect individual shared objects instead of memory pages, the minimum unit of MPK protection, potentially containing multiple shared objects. Third, KARD must concurrently monitor many threads and objects, even though MPK only provides a limited number of protection keys.

To overcome these challenges, KARD implements (a) an automated scheme that identifies shared objects accessed in critical sections by trapping access on them using memory protection (§5.3), (b) a consolidated unique page allocation scheme that assigns unique virtual pages to each memory object and ensures a low memory footprint using virtual page consolidation techniques (§5.3), and (c) an effective key assignment scheme that reuses keys within critical sections to ensure that KARD seldom runs out of protection keys (§5.4).

Furthermore, KARD dynamically analyzes all MPK-driven access violations (General Protection Fault (#GP)) raised by the program to prune out redundant or spurious faults. In particular, KARD uses a new technique to prune spurious faults, *protection interleaving*, that can detect byte-level access made by each thread and test the validity of raised warnings (§5.5).

We implement KARD using an LLVM compiler pass to trap heap allocation and synchronization calls, and a C++-based runtime library that creates unique paged heap objects and assigns protection to executing threads (§6).

We evaluate KARD using the PARSEC and SPLASH-2x benchmarks [8] as well as four real-world applications: NGINX [41], memcached [17], pigz [5], and Aget [19]. Our evaluation shows that KARD's geometric mean of performance overhead (under 4 threads) is 7.0% while running the benchmarks and 5.3% while running the real-world applications (§7.2). Moreover, KARD detects all real-world data races involving inconsistent lock usages and incurred one false positive (§7.3). KARD is also readily scalable: 10 out of 15 benchmarks were slowed by less than 30% even under 32 threads (§7.4). While KARD incurs a high memory overhead (more than 200%) in a few benchmark applications, its memory overhead is mostly modest, with a geometric mean of 68.0% and 85.6%, under benchmark and real-world applications, respectively (§7.5).

2 BACKGROUND

2.1 Data Race Terminology

For the sake of clarity, this section concisely describes the data race terminology used in this paper.

Data race. A program has a data race [20] when two or more threads perform a data access to the same memory location, in such a way that they can be executed simultaneously on a multi-core machine, and at least one of the accesses is a write.

Table 1: Inconsistent lock usage (ILU) between concurrent accesses to the same object by two different threads t_1 and t_2 (✓: in scope, ✗: out of scope).

Concurrent access to the same object		ILU
t_1	t_2	
With lock l_a	With lock l_b	✓
With lock l_a	No lock	✓
No lock	With lock l_b	✓
No lock	No lock	✗

Lock. A lock is a programming abstraction that allows a thread to synchronize the execution of its code with other concurrent threads. At a lock acquisition site, a thread stalls its execution if the lock is exclusively held by another thread.

Critical section. A critical section is a code region between lock andtgo unlock functions. Even if the program can acquire different sets of locks on a given code region, we still consider it a single critical section.

Sharable object. A sharable object is any heap or global object in a program, accessible to any executing program thread.

Shared object. A shared object is any sharable object accessed within a critical section.

2.2 Intel Memory Protection Keys (MPK)

Intel MPK is a CPU extension for per-thread memory protection that assigns either no access, read-only, or read-write permissions to a group of memory pages [14]. MPK currently supports up to 16 different *protection keys* (k_0 to k_{15}), which are assigned to memory pages through `pkey_mprotect()` system calls. However, since the first key (k_0) is reserved for backward compatibility, the effective number of available keys is 15.

With MPK, each thread can configure its access to memory pages protected by a certain protection key via a thread-local register, PKRU. In particular, MPK provides two non-privileged instructions, RDPKRU and WRPKRU, to allow the thread to retrieve and update its PKRU, respectively. These instructions are fast: RDPKRU takes less than 1 cycle and WRPKRU takes around 20 cycles [43]. The main reason behind their efficiency is that they do not change page table entries or require a TLB flush unlike `mprotect()`.

3 MOTIVATION AND GOAL

This section explains the significance of inconsistent lock usage in real-world data races, explains the limitation of existing detection schemes, and defines our goal.

3.1 Inconsistent Lock Usage

Inconsistent lock usage (ILU) refers to scenarios where a program's threads concurrently access the same memory object using different locks or only some of the accesses are synchronized using a common lock [28]. Table 1 illustrates ILU's scope. Specifically, two threads

concurrently access the same shared object while holding different locks, l_a or l_b, or only one of them acquires either l_a or l_b.

ILU is inspired by the traditional lockset algorithm [49], but considers concurrency to ignore many scenarios where lockset reports false data race warnings. In particular, lockset assumes that if an access to a shared object is using a set of locks that is inconsistent (i.e., no common lock) with a previous access to the object, it can result in a data race. However, lockset is agnostic to whether the two accesses can concurrently occur or not. Hence, in practice, lockset falsely reports many instances that do not result in a data race.

In contrast to lockset's scope, ILU is aware of concurrently executing threads; thus, it is more precise in relation to instances that might result in a data race. However, ILU is schedule-sensitive, i.e., the threads must be scheduled in a way that results in inconsistent lock usage, unlike lockset. We believe that such a trade-off is essential because a schedule-sensitive scope mitigates instances of chasing false bugs in development and testing settings, which are especially undesirable [34, 48]. Furthermore, as long as the tool employing such a scope is lightweight, it can be utilized under many test cases to detect data races in diverse code regions, unlike the expensive schedule-insensitive lockset.

Data races due to ILU scenarios are very common in real-world applications: our analysis shows that the majority of data races, reported by TSan [52], in real-world applications are associated with ILU. We analyzed real-world data races found by TSan that were eventually *fixed* [58], to filter out potential benign or intentional data races. We randomly chose 100 fixed bug reports for which we could find attached TSan logfiles (stacktraces of execution), and manually investigated them. We confirmed that 69% of them were involved ILU; that is, at least one of the conflicting threads in each report was holding a lock.

Note that TSan might be biased in its reported data races. This potential bias reflects a general limitation of real-world application bug studies caused by the difficultly to analyze the entire population of both identified and unidentified bugs [22, 24]. We chose TSan because it is the state-of-the-art in dynamic data race detection and continuously finds critical bugs in real-world applications. Furthermore, TSan maintains a robust and verifiable bug database. In this regard, our analysis on TSan's reported bugs empirically shows that many real-world data race bugs can be classified under ILU.

3.2 Limitations of Existing Approaches

Existing systems [9, 13, 18, 20, 37, 38, 42, 52, 62, 64, 65] detect data races due to ILU or more, but have various performance, deployment, usability, or scope limitations. Table 2 illustrates the characteristics of existing approaches.

Expensive memory instrumentation. Systems like TSan [52] and TxRace [62] employ compiler instrumentation of all or partial memory accesses, allowing them to report a large class of data races, but with prohibitive performance costs (i.e., 7× for TSan and 5× for TxRace under 4 threads). A recent study [34] shows that such overheads hinder their adoption in many development or testing scenarios.

Table 2: Comparison between KARD and existing approaches. Requirements include expensive memory instrumentation (MI), system (software or hardware) change (SC), and developer effort (DE) (●: required, ○: not required). Scope includes inconsistent lock usage (ILU) and inconsistent lock usage and others (ILU+).

System	Requirements			Scope	Overhead
	MI	SC	DE		
Compiler MI					
Eraser [49]	●	○	○	ILU	Very high
Inspector XE [12]	●	○	○	ILU+	Very high
TSan [52]	●	○	○	ILU+	Very high
Valor [9]	●	○	○	ILU+	High
Custom Hardware					
HARD [65]	○	●	○	ILU	Low
Conflict Exception [38]	○	●	○	ILU+	Low
Probabilistic Sampling					
DataCollider [20]	○	○	○	Sampled (ILU+)	Low/moderate
Pacer [13]	●	○	○	Sampled (ILU+)	Moderate/high
Memory Protection					
Aikido [42]	○	●	○	ILU+	Very high
PUSh [64]	○	●	●	ILU	Low
KARD (our work)	○	○	○	ILU	Low

System changes. Other schemes improve the performance of race detection by proposing system changes in the form of (a) hardware modifications [38, 65] or (b) software modifications to the OS [64] or hypervisor [42]. These suffer from deployment problems because custom hardware is nontrivial to manufacture and system software changes are less feasible in many environments (e.g., VM or container instances).

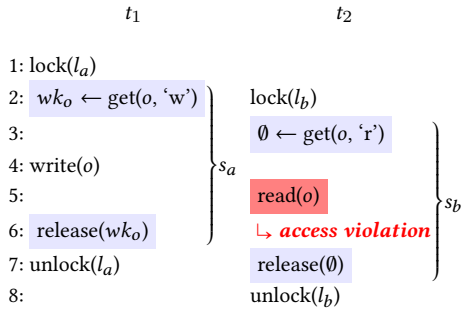
Developer efforts. Another approach to improve data race detection performance is selectively monitoring shared objects manually annotated by developers. However, given the challenges of manual annotation, adoption of such schemes is less likely to occur in practice. For example, PUSh [64], a scheme involving such developer effort, required 35 hours for annotation on a benchmark application (i.e., streamcluster [8]).

Probabilistic protection scope. Sampling techniques improve the performance of data race detectors by monitoring a small subset of instructions [13] or objects [20] at a time. However, low sampling rates for good performance yield low probabilistic effectiveness.

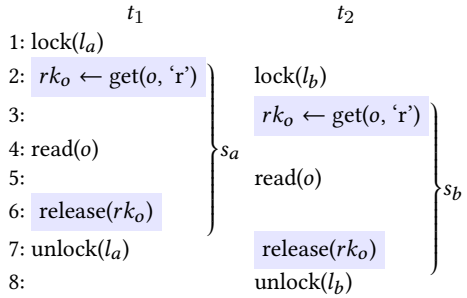
3.3 Goal

Motivated by the significance of data races due to ILU (§3.1) and the limitations of existing work (§3.2), we intend to design a novel data race detector, specifically for ILU data races, that satisfies the following goals.

- **Lightweight:** No individual memory access instrumentation
- **Deployable:** Neither hardware nor system software changes
- **Automated:** No manual annotation
- **Systematic:** No sampling



(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.

Figure 1: Example of key-enforced access during inconsistent lock usage between two threads t_1 and t_2 on a memory object o.

4 KEY-ENFORCED RACE DETECTION

This section introduces an algorithm based on the concept of *key-enforced access* to detect data races due to ILU. With key-enforced access, each shared object accessed in critical sections is protected by a key. A thread that enters a critical section can acquire a key to access a shared object and, later, the thread releases the acquired key when it exits the critical section.

Key-enforced access further distinguishes read-only keys from read-write keys to fulfill the concept of *shared read* and *exclusive write*. In particular, a thread can acquire a read-only key for an object o, rk_o, only if no other thread is holding a read-write key for o, wk_o. In contrast, a thread can acquire wk_o only if no other thread is holding wk_o or rk_o. Any read from o without rk_o or wk_o and any write to o without wk_o imply unordered memory access—the access can conflict with some other threads' concurrent access on o, resulting in a data race.

Example. Figure 1 depicts how key-enforced access works for exclusive write and shared read. Two threads, t_1 and t_2, access a shared object, o, within their critical sections, s_a and s_b, by obtaining two independent locks, l_a and l_b, respectively (i.e., under ILU).

Figure 1a depicts an instance of exclusive write. In this example, t_1 executes first and acquires a read-write key for o, wk_o (line 2). Then, t_2 requests the read-only key, rk_o, but fails to acquire it because t_1 holds wk_o (line 3). Hence, when t_2 attempts to read from o, it results in an access violation (lines 5–6).

Algorithm 1: Key-enforced race detection algorithm.

***t*:** thread
***s*:** critical section
***o*:** sharable object
***rk_o*:** read-only key for o
***wk_o*:** read-write key for o
***$K(t)$*:** set of keys held by t
***$K_R(s)$*:** set of keys for s with read-only permissions
***$K_W(s)$*:** set of keys for s with read-write permissions
***K_R*:** set of keys held with read-only permissions
***K_F*:** set of keys no thread holds

```
1  while t is running do
2      if t enters s then
3          push(K(t))
4          K(t) ← K(t) ∪ (K_R(s) ∩ (K_F ∪ K_R)) ∪ (K_W(s) ∩ K_F)
5          K_R ← K_R ∪ (K_R(s) ∩ K_F)
6          K_F ← K_F − (K(t) ∩ K_W(s))
7      if t exits s then
8          K_F ← K_F ∪ (K(t) ∩ K_W(s))
9          K(t) ← pop()
10     if t reads from o and (rk_o or wk_o) ∉ K(t) then
11         if wk_o ∉ K_F then
12             log potential race
13         else if t is executing s then
14             K(t) ← K(t) ∪ {rk_o}
15             K_R ← K_R ∪ {rk_o}
16             K_F ← K_F − {rk_o}
17             if wk_o ∉ K_W(s) then
18                 K_R(s) ← K_R(s) ∪ {rk_o}
19     if t writes to o and wk_o ∉ K(t) then
20         if (wk_o ∉ K_F) or (rk_o ∉ (K_F ∪ K_R)) then
21             log potential race
22         else if t is executing s then
23             K(t) ← K(t) ∪ {wk_o}
24             K_F ← K_F − {rk_o, wk_o}
25             K_W(s) ← K_W(s) ∪ {wk_o}
26             K_R(s) ← K_R(s) − {rk_o}
```

Figure 1b depicts an instance of shared read. Here, t_1 executes first but acquires rk_o (line 2). Therefore, t_2 can successfully acquire rk_o as well (line 3) and no violation is reported.

Algorithm. We design an algorithm for key-enforced access (algorithm 1). This algorithm maintains five types of key sets for each thread, each critical section, or the entire execution: (a) $K(t)$ is a set of keys that a thread t is currently holding; (b) $K_R(s)$ is a set of keys that a critical section s needs with read-only permissions; (c) $K_W(s)$ is a set of keys that a critical section s needs with read-write permissions; (d) K_F is a set of keys that no thread is currently holding, i.e., a set of free keys; (e) K_R is a set of keys that some threads have acquired with read-only permissions. In the beginning, K_F is initialized with a set of all keys, while the remaining sets are $\emptyset$. Our algorithm updates $K(t)$ whenever a thread t acquires or

releases a key k, and updates $K_R(s)$ and $K_W(s)$ according to the keys acquired while executing s.

If a thread t enters a critical section s, our algorithm first backs up its current $K(t)$ (lines 2–3). Then, t is given a subset of $K_R(s)$ that no thread holds with read-write permission and a subset of $K_W(s)$ that no other threads currently hold (line 4). Furthermore, it updates K_F to reflect that t exclusively acquired some keys, while updating K_R to show that the thread acquired some keys with read-only permissions (lines 5–6). Then, when t exits s, our algorithm releases the keys that t acquired either at the start of or during the execution of s (lines 7–9).

If t attempts to read from o without rk_o or wk_o (line 10), our algorithm checks whether another thread holds wk_o (line 11). If some thread t_* holds wk_o, it implies that t_* can write to o any time before or after t reads o. Therefore, this is a potential data race (line 12). If no thread holds wk_o and t is executing a critical section s, it claims rk_o to prevent a concurrent write from another thread (lines 13–14). Furthermore, to track that rk_o is held by some thread, the algorithm adds the key to K_R and removes it from K_F (lines 15–16). Also, if wk_o is not in the scope of the critical section s, the algorithm includes rk_o into $K_R(s)$ (lines 17–18).

On the other hand, if t attempts to write to o without wk_o (line 19), our algorithm checks whether any other thread t_* holds wk_o or rk_o (line 20). If this holds, our algorithm treats the access as a potential data race (line 21). If the key is free and t is executing a critical section s, the thread claims wk_o by adding it to $K(t)$ and $K_W(s)$ while removing it from $K_R(s)$, K_R, and K_F (lines 22–26).

5 KARD DESIGN

5.1 Overview

We design KARD, a practical dynamic data race detector that realizes our key-enforced race detection algorithm (§4) with a commodity per-thread memory protection mechanism, MPK (§2.2). KARD leverages MPK's per-thread read-only and read-write permissions on memory pages to protect sharable objects, accessed in critical sections, and to fulfill the requirements of key-enforced access.

Initially, KARD defines memory protection domains to classify sharable objects (i.e., heap and global variables) according to whether they are accessed in critical sections and whether those accesses were read or write (§5.2). To enable these domains, KARD implements (a) a consolidated unique page memory allocator to assign each sharable object unique virtual pages, allowing independent protection of each object using MPK and (b) an on-demand sharable object tracking scheme to accurately place objects in their respective domains (§5.3).

During program execution, KARD identifies new shared objects and enforces the protection domains for threads during critical section execution by carefully assigning protection keys to threads within distinct critical sections (§5.4).

Lastly, on access violations of protection domains, KARD performs automatic analysis to filter out redundant or non-racy violations, including initialization event detection, protection interleaving, and faulting address-based pruning (§5.5).

5.2 Memory Protection Domains

KARD classifies a program's sharable objects into three domains: READ-ONLY, READ-WRITE, and NOT-ACCESSED. The domains are enforced with different protection keys to ensure different access semantics on each domain during execution.

READ-ONLY domain. Each shared object that has been only *read* within critical sections belongs to the READ-ONLY domain protected using k_{ro} (k_{14}). In KARD, all threads (whether in critical or non-critical sections) have k_{ro} with a *read-only* permission, so they can always read these objects. However, write access on it is prohibited for domain migration or data race detection (§5.5).

READ-WRITE domain. Each shared object that has been *written* at lease once within critical sections belongs to the READ-WRITE domain, and is protected using one of 13 protection keys (k_1 to k_{13}). These protection keys are only provided to a thread that is executing a critical section, with either *read-only* or *read-write* permissions. To allow shared reads, multiple threads can obtain the same READ-WRITE domain key with read-only permission. However, to ensure exclusive write, only a single thread can obtain a READ-WRITE domain key with read-write permission at a time.

NOT-ACCESSED domain. Each newly created sharable object belongs to the NOT-ACCESSED domain, and if it is accessed within a critical section, it is migrated to either READ-ONLY or READ-WRITE domain. KARD protects new sharable objects using k_{na} (k_{15}), and retracts k_{na} from threads during their critical section execution. This allows KARD to determine if a thread accessed a newly created sharable object within its critical section via an access violation.

Finally, all memory regions must have a protection key when MPK is enabled. Therefore, KARD protects memory objects that are not sharable (e.g., thread-local variables) or should always be accessible to all threads (e.g., mutex variables) using the default key, k_{def} (k_0 in current Intel CPUs).

5.3 Sharable Object Tracking

KARD tracks sharable objects to identify which ones are shared (i.e., read or written in critical sections) and changes their protection domains using MPK. However, this presents two challenges. First, MPK works at the granularity of pages (typically 4 KiB) but native allocators might create many objects in a single page. Second, accurately determining whether an object is shared and identifying the access type (i.e., read or write), without comprehensive memory access instrumentation, is non-trivial. To overcome these challenges, KARD implements consolidated unique page allocation and an automated shared object identification scheme.

Consolidated unique page allocation. Existing heap allocators store multiple objects in the same page to reduce memory consumption, which is incompatible with KARD's protection. Since objects stored in the same page can belong to different protection domains, such compact allocation would result in extraneous access violations. This hampers both accuracy and performance of KARD.

To ensure per-object protection of heap objects, KARD replaces a program's heap allocation routines (e.g., `malloc()` and `new()`) with its custom memory allocation routines that assign unique virtual pages to each object. In addition, KARD maintains the metadata (i.e.,

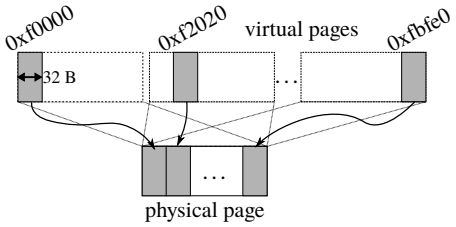


Figure 2: Consolidated unique page allocation. 128 unique virtual pages of 32 B objects mapped into a single physical page.

base address and size) of each allocated object to locate the corresponding object given any (faulting) address. However, assigning unique pages to each object can waste a considerable amount of physical memory, especially if a program has many small objects.

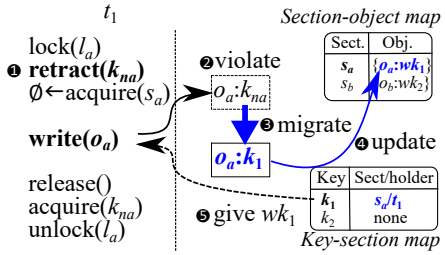
KARD conserves physical memory by consolidating different, small objects into a single physical page via shared mapping. Specifically, KARD first creates an in-memory file using `memfd_create()`. Then, whenever the program tries to allocate some memory, KARD invokes `mmap()` (with the `MAP_SHARED` flag) to create a new virtual page mapped into the in-memory file. If multiple allocations are mapped into the same physical page of the in-memory file (e.g., Figure 2), KARD shifts each allocation's page-internal base addresses and returns these shifted addresses to ensure that different allocations do not overlap within the physical page (like minipage or page aliasing techniques [16, 25, 29, 44, 45]). In addition, KARD increases or decreases the in-memory file's size by invoking `ftruncate()` according to a program's memory requirements.

Like heap objects, KARD assigns each global object to unique virtual pages to ensure per-object protection. To accurately maintain metadata for global objects, KARD aggregates each global object's information (e.g., address and size) during compilation, and provides this information to its run-time library by inserting function calls at the beginning of the program (refer to §6).

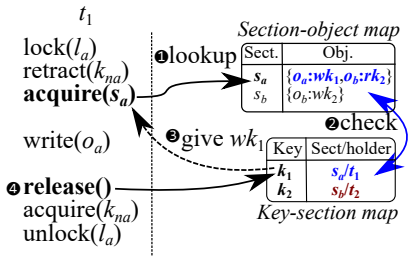
Automated shared object identification. KARD progressively identifies shared objects, from the list of all sharable objects, whenever critical sections execute. In particular, KARD traps every critical section execution with compiler instrumentation and prevents threads executing critical sections from accessing sharable objects not previously identified as shared. Hence, if the thread accesses such an object, a `#GP` will occur.

KARD traps critical section entries and exits by replacing synchronization calls, e.g., `lock()` and `unlock()`, with corresponding wrapper functions. Also, KARD provides the virtual address of a synchronization call site to the wrapper function to differentiate between critical sections during execution.

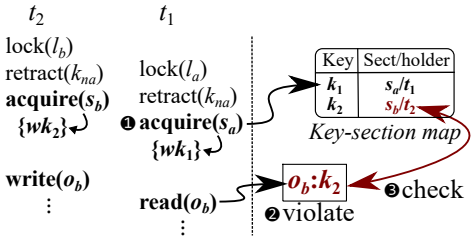
Figure 3a illustrates shared object identification during critical section execution. When a thread t_1 enters a critical section, KARD revokes the access of t_1 to k_{na} (❶). Thus, a protection fault (`#GP`) related to k_{na} implies that t_1 accessed a sharable object o_a in the NOT-ACCESSED domain (❷). Using the faulting address and access type (i.e., read or write) from the `#GP`, KARD migrates the object o_a to either the READ-ONLY (i.e., k_{ro}) or READ-WRITE (i.e., one of k_1 to k_{13}) domain according to the access type (❸). Then, KARD updates an internal data structure, the *section-object map* (❹), that tracks



(a) Object tracking. Identify the first access on an object o_a by a thread t_1 executing a critical section s_a and migrate o_a to a corresponding domain.



(b) Domain enforcement. Attempt to acquire the keys of objects belonging to s_a when t_1 enters it.



(c) Race detection. Analyze a violated memory access on o_b based on the current key holding threads.

Figure 3: Different KARD stages to (a) progressively identify shared objects, (b) acquire the keys of objects within critical sections, and (c) detect potential data races on objects. All these stages occur continuously during the same program execution.

the shared objects accessed in each critical section and is used for domain enforcement. Lastly, t_1 acquires the key protecting o_a, if it did not hold the key before (❺) (refer to §5.4).

Similarly, if a critical section raises a `#GP` because it attempts to write to an object belonging to the READ-ONLY domain, KARD migrates the object to the READ-WRITE domain.

5.4 Domain Enforcement

While progressively identifying shared objects (§5.3), in the same execution, KARD enforces protection domains during critical section executions to detect data races using *key-enforced access* (§4). Each identified shared object (§5.3) is assigned a protection key, and a thread either proactively (at a critical section entry) or reactively (during critical section execution) acquires the key. When a thread exits a critical section, it releases the keys acquired within the critical section.

Effective key assignment. KARD uses several rules to smartly assign keys to identified shared objects. Ideally, each shared object (in the READ-WRITE domain) should be assigned a unique key to enforce different access semantics for each critical section. However, due to MPK's hardware limitation, KARD has a limited number of keys for objects migrated to the READ-WRITE domain (i.e., 13).

KARD follows three rules to effectively use protection keys. First, if a faulting thread holds one or more protection keys, KARD assigns one of the held protection keys to the object (using `pkey_mprotect()`). Second, if the faulting thread does not hold any protection key (e.g., the first write access on a sharable object in a critical section), KARD looks for an unassigned protection key, protects the object with the key, and allows the thread to acquire the key.

Third, if all protection keys are assigned, KARD either (a) *recycles* an assigned key that no thread is currently holding or (b) *shares* a key between threads if all keys are currently held. Note that recycling is preferable since KARD can move the objects that have been protected by the recycled key to the READ-ONLY domain. This allows KARD to trap potential data races (through writes) on the recycled objects. Thus, recycling neither harms KARD's accuracy nor preciseness, but only slows down the program due to repeated domain migration. In contrast, sharing can result in false negatives (refer to §7.3). Nevertheless, our evaluation shows that both recycling and sharing is rare (§7.3); hence, in practice, their impact is minor.

Proactive key acquisition. With KARD, a thread will proactively acquire keys, belonging to already identified objects, when it enters a critical section. In particular, protection keys are acquired at critical section entries using a user-level instruction, `WRPKRU`. Note that key acquisition can be racy; thus, KARD employs internal synchronization (i.e., atomic operations), like general lock functions, to prevent such problems.

Figure 3b illustrates how a thread acquires keys at a critical section entry. When a thread enters a critical section, KARD looks up the *section-object map* to find the shared objects accessible in the critical section and associated keys (❶). Then, KARD checks the *key-section map*, a data structure that maintains which sections (and threads) currently hold what keys, to determine which keys are available (❷). The thread can acquire an available key with (a) read-write permission if no other thread holds the key or (b) read-only permission if no other thread holds the key with read-write permission. In the figure, a thread t_1 acquires k_1 with read-write permission, but fails to acquire k_2 because another thread t_2 is holding it (❸). Lastly, KARD pushes the thread's current keys in a thread-local stack to restore when it exits the critical section.

Reactive key acquisition. At some object identifications (§5.3), a thread must reactively acquire a key assigned to a newly identified shared object, if the thread did not acquire that key at critical section entry. This reactive key assignment is performed by KARD's fault handler (explained in §5.5). The fault handler, however, cannot use `WRPKRU` for this key assignment because the program thread is not currently running. Instead, the fault handler modifies the thread's stored process context for the `PKRU` register [43, 57] to allow the thread to possess the key when it is scheduled again.

Key release. When a thread exits a critical section, the thread releases protection keys acquired reactively or proactively (Figure 3b-❹). More specifically, if a thread exits to a non-critical section, it releases all of its protection keys (k_1 to k_{13}, except k_{ro}) and acquires k_{na}. On the other hand, if the thread exits a *nested* critical section (i.e., it is still within a parent critical section), it reverts back to its old keys from the thread-local stack. Lastly, KARD timestamps (using `RDTSCP`) each time a thread releases protection keys to consider a fault handling delay during data race analysis (refer to §5.5).

5.5 Race Detection and Filtration

KARD redirects all protection faults (`#GP`s) to its custom fault handler to detect potential data races while handling missing or spurious violations due to fault handling delay and initialization events. Furthermore, KARD employs a *protection-interleaving* scheme to effectively filter out false positives. Lastly, KARD automatically prunes redundant violations.

Protection fault (`#GP`) redirection. KARD registers a custom fault handler to trap all MPK-based `#GP`s and obtain information related to the faulting address, including protection key, access type, and process context (e.g., instruction pointer).

The system raises a `#GP` for attempted access on (a) sharable object belonging to the NOT-ACCESSED domain (protected by k_{na}), (b) shared object protected with a key that the thread did not acquire, or (c) shared object protected with a key that the thread acquired with read-only permissions. Considering (a), KARD assigns a protection key to the object (refer to §5.4). The remaining scenarios, (b) and (c), could be data races but require further analysis to check whether the faulted key is held by another thread with read-write permissions.

Figure 3c illustrates a potential data race scenario. In this example, we assume that t_2 executes first and acquires a protection key, wk_2, to write in an object o_b. Then, when t_1 executes, it fails to acquire a key rk_2 because t_2 is holding wk_2, according to the key-section map (❶). Hence, when t_1 tries to read o_b, it raises a fault (❷). KARD checks whether the key protecting o_b is currently held by any thread (❸), and confirms there is a data race between t_1 and t_2.

Note that KARD uses timestamps to assess whether a protection key was held when the `#GP` occurred because the key might be released by the time the handler is invoked. KARD checks whether the time difference between key release and fault handler invocation is below the average fault handling delay (e.g., 24,000 cycles on our machine whose specifications are described in §7.1).

Protection interleaving. With KARD, a thread proactively acquires keys at critical section entries and a single key protects the entire object to minimize runtime overhead and efficiently use a limited number of protection keys, respectively. This, however, potentially introduces false positives due to (a) conditional branches in critical sections and (b) concurrent access to different offsets in an object (detailed analysis in §7.3).

KARD mitigates false positives using a *protection-interleaving* scheme that alternates protection keys assigned to faulted objects to further test the accuracy of a raised protection fault. Figure 4 illustrates protection interleaving between two threads t_1 and t_2. In particular, t_2 raises a `#GP` for an object o that is protected by a key k_1, held by t_1 (line 6). KARD *forcefully* protects o with one of the keys held by t_2 (or protects o with a free key and assigns the key to t_2), k_2, and allows t_2 to proceed (line 7). Later, if t_1 accesses o, it will raise another `#GP` (line 8). Thus, KARD observes

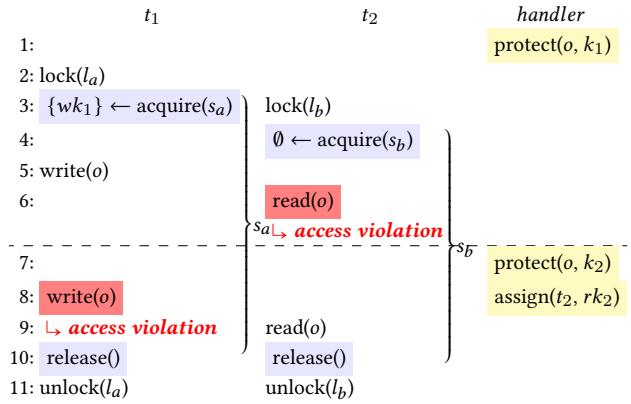


Figure 4: Protection interleaving to observe access violations to the same object o from different threads. The handler changes a protection key assigned to o (from k_1 to k_2) trigger access violations.

multiple #GPs on the same object from different critical sections to test whether the threads are concurrently accessing the memory object at the same offset. If KARD finds out that the object is accessed at different offsets, it removes the data race record from its internal log. Lastly, KARD terminates protection interleaving to proceed the program's execution by temporarily not protecting the object until all conflicting threads exit their critical sections.

The effectiveness of protection interleaving depends on the critical section size and memory access order. Specifically, small critical sections can terminate before protection is interleaved and #GPs might not re-occur depending on memory access order. These limitations can be mitigated with delay injection and multiple runs.

Automated pruning. KARD automatically prunes redundant and non-racy violations using its metadata (e.g., section-object map). It prunes (a) redundant faults of the same object at the same offset from different threads and (b) spurious faults due to access at different offsets of the same object (captured by protection interleaving).

Potential data race record. Based on the filtered #GP and metadata, KARD constructs potential data race information including: (a) both sections that raise the fault and holds the key, (b) the faulted object, (c) the faulting thread's access type, (d) thread identifiers and process contexts, and (e) timestamp.

6 IMPLEMENTATION

We implemented KARD using the LLVM 7 compiler suite [32]. Our implementation consists of backend LLVM function and module passes, as well as a runtime library. The runtime library has three main components: (a) a custom allocator along with wrapper functions for standard heap allocation, (b) wrapper functions for POSIX, Pigz [5], and NGINX [41] synchronization functions, and (c) a fault handler. Our backend code consists of 966 Lines of Code (LoC) and the runtime library consists of 1,888 LoC written in C/C++, counted using `sloccount` [2].

For simple management, KARD's custom allocator uses standard C++ data structure libraries to manage allocated memory objects and returns a multiple of 32 B to each memory allocation request. Currently, KARD invokes `mmap()` for each memory allocation, so it can suffer from high performance overhead if an application frequently allocates memory objects. However, such applications are rare (refer to §7.2). In the future, to reduce the overhead of frequent `mmap()` calls, KARD can recycle de-allocated virtual pages [64]. Moreover, KARD does not consolidate page-aligned global variables, thereby increasing the overall memory consumption of the system. We confirm that this does not affect the runtime performance of our tests, and KARD's reported memory overhead is over-estimated rather than under-estimated.

KARD does not currently cover non-locking synchronization primitives including ad-hoc ones. Generally, ad-hoc synchronization primitives are considered harmful [59]. If such synchronization must be employed, KARD's backend can be updated either using annotations or static analysis [59] to trap their execution and enforce memory protection.

7 EVALUATION

This section evaluates KARD in terms of performance overhead, effectiveness, scalability, and memory consumption.

7.1 Experimental Setup

We conducted all experiments on a Dell Precision 7820 Workstation featuring two Intel Xeon Silver 4110 processors where each processor consists of 16 logical cores at 2.1 GHz (32 logical cores in total), and 32 GiB of memory. The workstation runs Ubuntu 18.04.2 LTS whose kernel version was 4.15.

7.2 Performance

This section illustrates KARD's performance overhead while executing a diverse set of benchmark and real-world applications. In general, there are three major factors affecting KARD's performance: (a) the number of protected sharable objects, (b) the number of critical section entries, and (c) data TLB (dTLB) miss rate. First, the number of `pkey_mprotect()` system call invocations linearly depends on the number of sharable objects (invoked at object allocation and migration). Second, KARD traverses and updates the *section-object* and *key-section maps* to change the protection keys assigned to threads whenever they enter critical sections. Lastly, since KARD's memory allocator assigns each sharable object unique virtual page(s), it consumes more virtual address space, hence, programs can suffer additional dTLB misses.

Settings and notation. We compare KARD's performance against (a) applications without data race detection (referred to as Baseline), (b) applications using KARD's memory allocator but no race detection (referred to as Alloc), and (c) applications with TSan (referred to as TSan). Alloc illustrates the impact of KARD's memory allocator on its overall performance. For Baseline and TSan, the applications were built using unmodified `clang-7`. We used the same compiler flags (except for race detection flags) for all four cases.

We ran most of the experiments using 4 threads. Despite being a modest configuration for production, 4 threads are suitable for testing, which is KARD's main focus. During testing, the developer controls the number of program threads and generally tries to simplify the testing scenario to optimize resource usage and simplify failure diagnosis. Note that a well-known data race study [36] shows

Table 3: The execution statistics and performance overhead of KARD on PARSEC, SPLASH-2x, and real-world applications with 4 threads. For execution time, peak memory, and dTLB miss rate, each of Alloc, KARD, and TSan show the added overhead (in %) over Baseline execution.

Bench type	Sharable objects		Shared objects		Critical sections			Execution time				Peak memory (RSS)		dTLB miss rate		
	Heap	Global	RO	RW	Total	Active	Entry	Baseline (s)	Alloc	KARD	TSan	Baseline (B)	KARD	Baseline	Alloc	KARD
PARSEC																
streamcluster	1,818	20	0	1	6	3	115,760	4.96	0.1%	0.3%	2264.7%	12,592	6.1%	0.00013	5.1%	9.2%
x264	15	420	0	0	2	2	33,521	1.749	0.4%	3.0%	485.3%	29,732	2.0%	0.00020	0.6%	2.6%
vips	102	3,933	377	213	5	2	37	2.145	0.6%	1.3%	889.8%	24,360	3.3%	0.00042	0.7%	3.8%
bodytrack	8,717	125	7	48	8	1	56,196	3.268	4.1%	10.4%	655.6%	20,224	123.2%	0.00003	21.9%	55.2%
fluidanimate	135,438	25	24	5	8	4	4,402,000	3.251	19.6%	61.9%	1222.3%	374,760	142.6%	0.00018	32.3%	72.0%
SPLASH-2x																
ocean_cp	370	30	2	2	24	2	6,664	3.803	-8.3%	-5.9%	911.4%	913,048	0.3%	0.00030	0.2%	0.4%
ocean_ncp	16	38	0	4	23	2	6,504	5.631	0.0%	0.0%	1036.2%	922,128	0.3%	0.01149	0.0%	0.0%
raytrace	6	60	1	2	8	3	986,046	4.355	1.3%	3.7%	1368.6%	7,712	28.5%	0.00002	0.3%	0.5%
water_nsquared	128,007	87	96,000	2	17	4	96,148	10.022	9.1%	18.0%	698.0%	12,260	4145.9%	0.00001	587.3%	890.2%
water_spatial	37,148	99	1	1	2	2	675	3.259	2.9%	5.6%	546.1%	25,324	516.9%	0.00004	147.1%	172.6%
radix	17	13	2	1	13	4	103	5.173	-1.4%	-1.0%	187.4%	1,051,536	0.2%	0.00407	0.1%	0.1%
lu_ncb	12	11	2	1	6	2	1,040	3.917	-5.7%	-5.2%	292.9%	34,952	5.9%	0.00049	-3.7%	-3.4%
lu_cb	26	10	0	3	6	2	2,080	3.517	-7.8%	-4.7%	259.0%	35,092	6.1%	0.00003	1.4%	2.3%
barnes	44	54	11	13	5	5	1,784,848	5.126	2.9%	34.1%	1582.9%	68,000	3.3%	0.00011	3.0%	37.1%
fft	11	26	14	1	8	2	32	2.874	0.7%	1.0%	265.1%	789,588	0.3%	0.00092	-0.2%	-0.2%
GEOMEAN									**1.0%**	**7.0%**	**690.9%**		**68.0%**		**25.3%**	**37.2%**
NGINX	500,007	461	0	100,002	26	3	200,008	15.144	13.3%	15.1%	258.9%	5,812	202.1%	0.00145	51.9%	65.2%
memcached	6,985	107	24	62	121	13	161,992	2.009	0.0%	0.1%	45.7%	5,892	31.8%	0.00110	9.6%	18.2%
pigz	861	53	7	10	10	5	45,782	0.254	2.9%	5.1%	229.9%	5,368	52.5%	0.00028	31.4%	71.2%
Aget	24	10	0	1	2	1	56,196	0.944	0.6%	1.4%	464.3%	2,468	95.3%	0.00294	3.7%	12.3%
GEOMEAN									**4.1%**	**5.3%**	**189.5%**		**85.6%**		**22.7%**	**39.2%**

that even 2 threads can manifest more than 95% of data race bugs; hence, 4 threads are appropriate for most data race testing scenarios.

Benchmarking suites: PARSEC and SPLASH-2x. We evaluated KARD with the PARSEC/SPLASH-2x benchmark version 3.0-beta-20150206 [8]. We ran all benchmarking applications that use locks and do not require code-rewriting to build using `clang-7`. We decided to omit benchmarks that do not use locks because they have no overhead under KARD and that require code rewriting because it can affect the benchmark's intended behavior.

Results. Table 3 shows that KARD incurs a low performance overhead with a geometric mean, compared to Baseline, of only 7.0%. Most of the benchmark applications incurred a very low overhead except a few that frequently enter critical sections or have many protected sharable objects. Especially, the two benchmarking tests, *fluidanimate* and *barnes*, that entered critical sections around 4.4 million and 1.8 million times within 3.3 s, respectively, incurred a higher overhead than the average. We conjecture that such behavior is more congenial to benchmark tests, and we did not observe this in our real-world applications (e.g., NGINX entered critical sections only around 200,000 times within 15.2 s in our evaluation). Besides, despite fewer critical section entries, *water_nsquared* has many protected shared objects; hence, it suffered from a high dTLB pressure and showed a high performance overhead.

Interestingly, applications with few sharable objects (e.g., *lu_cb* and *ocean_cp*) ran slightly faster with KARD, compared to Baseline. The reason is that KARD's memory allocator trades off simplicity for memory consumption (§7.5) and, therefore, we observe the same trend with Alloc. Importantly, even if we compare KARD's performance on each benchmarking application against the best performance of either Baseline or Alloc, the overall geometric mean of KARD's performance overhead only rises to 8.7% from 7.0%.

Lastly, KARD executed approximately two orders of magnitude faster than TSan, under 4 threads, because KARD does not employ expensive compiler instrumentation of memory accesses. Nevertheless, TSan covers more diverse data race types than KARD (refer to Table 2).

Real-world applications. We evaluated KARD with four real-world applications: web server (NGINX 1.17.1 [41]), key-value store (memcached 1.5.16 [17]), compression software (pigz 2.2.4 [5]), and download manager (Aget 0.4.1 [19]). The geometric mean of KARD's overhead, under 4 threads, was 5.3% for these applications, compared to Baseline.

Web server—NGINX. Web servers employ multi-threading to serve multiple clients simultaneously with low latency. We used ApacheBench (`ab`) [56] to send 100,000 requests through 100 concurrent clients to an NGINX web server. We ran four different tests to retrieve files of sizes of 128 kB, 256 kB, 512 kB, and 1 MB, from the NGINX web server. We chose the file sizes after consulting existing web server traffic reports [6]. The average latency overhead across the tests was 15.1%. In particular, we observed that the overhead was higher when the file size is smaller. For example, the `ab` tests for 128 kB file showed an overhead of 58.7% while the overhead for the 1 MB file was 8.8%. Thus, the additional I/O overhead of larger files amortized KARD's runtime overhead.

Key-value store—memcached. Concurrency is a large enabler of modern key-value stores. We evaluated KARD against memcached [17], a popular key-value store, using a benchmarking tool, `twemperf` [47]. We configured `twemperf` to send 20,000 `set` requests of 1 B size through 2,000 concurrent connections. The overhead

Table 4: Potential false negative and positive scenarios for KARD, and how KARD mitigates them.

Potential issue	Mitigation
False negative	
Sharing protection keys	Share keys amongst disjoint sections
False positive	
Different offset in an object	Employ protection interleaving
Non-access in critical section	Employ protection interleaving

Table 5: Number of memcached threads versus the unique and concurrent critical sections (CS), recycling events, and sharing events.

Number of threads	4	8	16	32
Total executed CS	161,992	162,254	163,314	164,517
Uniquely executed CS	45	45	45	45
Maximum concurrent CS	13	14	16	16
Key recycling events	724	764	771	808
Key sharing events	11	22	30	116

was negligible. In particular, memcached accessed very few shared objects in its critical sections and had a modest number of critical section entries; therefore, the overhead remained low, despite a few key sharing and recycling events (refer to §7.3).

Other applications—Aget and pigz. Aget [19] accelerates file download using multiple threads. We used Aget to download a Linux kernel image (105 MB size). To account for network delays, we downloaded the file 100 times. KARD's slowdown was only 1.4%.

Another application, pigz [5], compresses or decompresses files using multiple threads. Our test comprised of decompressing the Linux kernel image (105 MB) 100 times using 4 concurrent threads. KARD's average overhead was 5.1%.

7.3 Effectiveness

This section analyzes KARD's effectiveness and presents results related to real-world data races reported by KARD.

Theoretical analysis. Table 4 provides an overview of the potential issues that KARD might face and various mitigation strategies adopted by KARD, in response.

False negatives. KARD can fail to capture data races involving ILU only under key sharing, since multiple threads can concurrently access objects protected by a shared key. KARD mitigates this problem by sharing keys amongst threads that are executing critical sections that do not access the same objects (by consulting the section-object map). However, KARD's mitigation will not be effective if all executing critical sections can access the same object. Nevertheless, our evaluation suggests that key sharing is *extremely rare* (explained below) due to KARD's effective key assignment. Furthermore, we envision improved memory protection hardware that provides a large number of keys. For example, researchers have proposed new hardware-software co-designs to efficiently support many protection keys [50, 60] that would effectively address KARD's false negatives.

False positives. KARD can report false positives because of the two reasons: (a) accessing different offsets in the same object or (b) protecting non-accessed objects.

First, KARD protects an entire object with a single key due to hardware restrictions (i.e., page-level protection and limited number of keys). Thus, it observes access violations even when two threads access different offsets of the same object. Second, KARD protects objects at critical section entries to avoid expensive memory access instrumentation. Therefore, it can over-protect a memory object if the thread executing a critical section does not access the object (e.g., due to conditional branches). KARD's analysis with protection interleaving (§5.5) can mitigate these false positives by accurately identifying the actual bytes of objects accessed by different threads in their respective critical sections.

Key sharing case study. To confirm whether false negatives due to key sharing can be a problem in practice, we ran all our benchmark and real-world applications with up to 32 threads (the maximum number of hardware threads our system supports §7.1). We found that only 1 out of 19 applications, memcached, required key sharing, but its sharing events were extremely rare: less than 0.07% of total executed critical sections.

In particular, memcached has 121 critical sections, based on our source code analysis, and executes many of them concurrently; hence, KARD had to employ key sharing. Table 5 shows the total number of critical sections executed, how many of the executed critical sections were unique or concurrent at one time, and corresponding sharing and recycling events, while increasing the number of threads.

Despite executing 13–16 concurrent critical sections, memcached required key sharing very few times, 11–116 (only 0.007%–0.07% of the total executed critical sections). While increasing the number of threads increases the active critical sections and produces more key sharing events, the difference is minimal. This shows that even for complex real-world applications with many critical sections and executing many threads, such as memcached, few critical sections execute concurrently; therefore, the limited number of keys provided by MPK should be sufficient in most cases.

We also observed a few key recycling events, 724–808 times (only 0.44%–0.49% of total executed critical sections). Note that recycling does not impair KARD's accuracy or preciseness (refer to §5.4).

We assume that a developer will use configurations that minimize key sharing events (i.e., a small number of threads) to avoid potential false negatives (Table 4). Even when configurations with high key sharing events (e.g., memcached with 32 threads) are employed, KARD can still probabilistically report a data race if it is triggered multiple times with different key sharing instances due to thread scheduling. Nevertheless, new hardware proposals or software fallback can avoid key sharing (refer to §8).

Real-world data races reported. KARD reported all real-world data race bugs that were also reported by TSan involving ILU with only 1 false positive (Table 6). Both KARD and TSan are schedule-sensitive; therefore, they might not produce the same number of data races on each run. However, we did not observe a variance for these applications.

Aget. KARD reported a single data race warning from Aget, similar to TSan. In particular, Aget writes to a single global variable (to

Table 6: Real-world data races reported by KARD and TSan.

Application	KARD	TSan	
		ILU	Non-ILU
Aget	1	1	0
memcached	3	3	0
NGINX	1	1	0
pigz	1	0	0

count the bytes downloaded by individual threads) within its critical section. KARD raised a warning because the variable is read outside of the critical section while another thread is updating it within a critical section. This data race has been previously reported [61].

memcached. KARD reported 3 data races involving ILU from memcached, which are also reported by TSan. The reported warnings belonged to two statistics collection heap variables and one global variable to track time. In particular, the heap variables were updated by a worker thread within its critical section and read by the main thread (outside its critical section). As for the global variable, its timing contents are updated by the main thread using a callback mechanism and read by another thread, within a critical section. We believe that this data race might be intentional since the main thread only updates the timing information.

NGINX. Both KARD and TSan reported a data race that originates from a racy heap access in a critical section during NGINX's initialization. This was the only data race warning reported by either of the tools, while executing NGINX.

pigz. KARD reported 1 false positive due to concurrent access at different offsets in a heap variable from pigz. Despite protection interleaving, KARD failed to prune this warning out because the corresponding critical section is too small such that KARD did not observe accesses from other threads. Therefore, it included the data race in the final report. Nevertheless, protection interleaving filtered out pigz's other access violations within relatively large critical sections.

7.4 Scalability

KARD maintains its performance efficiency for most of the benchmarking applications even with up to 32 threads. Figure 5 shows the scalability of KARD while executing PARSEC and SPLASH-2x applications. The geometric mean of KARD's performance overhead with 8, 16, and 32 threads, is 24.4%, 63.1%, and 107.2%, respectively. We observed that KARD scaled well for most benchmarks, except the three applications, *fluidanimate*, *water_nsquared*, and *barnes*, due to their large numbers of critical section entries and sharable objects. However, excluding such worst-case applications, the geometric mean of KARD's overhead was only 5.8%, 12.4%, and 19.0%, for 8, 16, and 32 threads, respectively.

7.5 Memory Consumption

KARD incurs memory overhead due to (a) unique page memory allocation and (b) internal metadata like the *section-object* and *key-section* maps. First, despite page consolidation, KARD can waste memory if the allocated object size is not a factor of the fixed virtual page consolidation size (currently set to 32 B). Also, KARD does

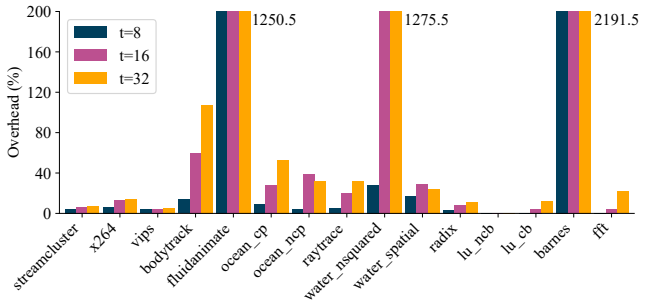


Figure 5: The scalability of KARD while executing PARSEC and SPLASH-2x with 8, 16, and 32 threads.

not currently consolidate the unique virtual pages of global objects (refer to §6); hence, it pays an additional memory cost that can be reduced in future implementations. Second, KARD consumes memory to maintain internal metadata, depending on the number of critical sections, sharable objects, and protection keys. Such memory consumption can be reduced by implementing optimized data structures for KARD instead of using standard C++ data structure libraries.

Despite the limitations of KARD's implementation, considering the evaluated applications, KARD's memory overhead is modest: a geometric mean of 68.0% for the benchmarking applications and 85.6% across our real-world applications, according to the Resident Set Size (RSS) [51] (Table 3). In particular, 10 out of 15 benchmarking applications have a memory overhead less than 10%. We observe that *water_nsquared* has the highest memory overhead. Upon closer inspection, we found that it allocates 128,000 of 24 B heap objects, making the unique page allocator waste memory (i.e., 8 B per-object since KARD allocates objects in multiples of 32 B). In contrast, although NGINX allocates many objects, its memory overhead is smaller than that of *water_nsquared* because the size of its objects is usually 32 B or 4 KiB. Thus, KARD can decrease the memory overhead by profiling the most common allocation size of a target program and using that size for allocation.

8 DISCUSSION

This section describes a potential problem that could occur with progressive shared object identification. Then, it presents ways to preserve KARD's accuracy even in key sharing scenarios using software and advanced hardware memory protection implementations. Lastly, the section discusses possible KARD extensions that would allow KARD to detect non-ILU data races and data races on program binaries without requiring source code instrumentation.

Progressive identification. Progressive shared object identification and domain enforcement (refer to §5.3) can potentially open a tiny time window that allows a thread outside of its critical section to access a shared object being identified within a different thread's critical section. KARD will not report such data races due to a lack of domain protection for the newly-identified shared object. However, we believe that such scenarios are extremely rare because the time window is so small and we did not observe them during the evaluation (refer to §7.3). Furthermore, KARD will report the data

race missed during identification if it occurs again during the same program execution. We leave such investigation to future work.

Software fallback and advanced hardware. KARD can overcome the limitations of MPK's current implementation using software per-thread memory protection mechanisms, such as those employed by iThreads [7], or advanced hardware implementations of per-thread memory protection. Since MPK provides only 16 protection keys, KARD must share protection keys in rare occasions (refer to §7.3). However, KARD's race detection algorithm is agnostic to the underlying memory protection mechanism, so it can revert to a software memory protection scheme [3, 10, 46] when it exhausts hardware protection keys. Nevertheless, software memory protection requires systems software changes and incurs significant performance costs (up to 100% [46]). In the future, KARD can be implemented on advanced hardware [50, 60] that allows up to 1000 protection keys to completely negate protection key limit issues.

Non-ILU extension. KARD can extend its race detection algorithm (refer to algorithm 1) to track data races that occur outside ILU's scope, by updating the algorithm to acquire protection keys for shared variables outside critical sections. However, on current MPK hardware, such an extension would not be effective because the limited protection keys would result in many instances of protection key sharing (which can result in false negatives). Nevertheless, software fallback and advanced hardware can circumvent protection key limitations (as explained before) to make such extension feasible in the future.

Program binary extension. KARD can act directly on unmodified program binaries by intercepting standard library calls for heap memory allocation and thread management, and using a custom loader to handle global variables. For example, KARD can intercept library calls via `LD_PRELOAD` to use its memory allocator and update per-thread protection domains while differentiating synchronization call sites using return addresses. Furthermore, for position independent binaries (compiled with `-fPIC`), KARD can change the locations of global variables to store them in unique virtual pages. We leave such implementation to future work.

9 RELATED WORK

This section provides an overview of data race detection schemes that are related to KARD. It initially describes schemes that implement dynamic data race detectors using software or hardware memory protection or use other commodity hardware features like transactional memory and processor monitoring units. The section concludes by providing a comparison between dynamic data race detectors like KARD and static data race detection techniques.

Memory protection for data race detection. Several prior studies use software memory protection to detect data races. MultiRace [45] uses minipages along with pointer swizzling to ensure per-thread view on a memory page, but requires expensive indirect memory access. ISOLATOR [46] and Abadi et al. [3] use a copy-on-write trick to ensure thread isolation. Whenever a thread writes a value into a shared read-only page, it will obtain a new private page copied from the original one. However, if the diverged pages have conflicting values, it is difficult to merge them to create a single view. Other schemes implement per-thread address spaces [10, 27, 35, 42] using kernel modifications. Such schemes invoke system calls and flush TLB entries to change per-thread permissions. Thus, they incur substantial performance overhead unlike KARD.

Virtual machine monitors have also been used to enable software memory protection for data race detection. In particular, SKI [23] uses a modified QEMU to analyze memory accesses at the virtual machine-level while ensuring schedule diversity. This allows SKI to detect kernel data races and other kernel concurrency bugs without modifying the tested system. Unlike SKI, KARD uses efficient hardware protection mechanisms and selectively analyzes memory accesses to ensure very low impact on application performance.

Like KARD, PUSh [64] uses MPK's efficient hardware memory protection to detect data races. However, KARD has three main technical contributions over PUSh. First, KARD proposes a key-based race detection algorithm that reports ILU data races using protection keys. Unlike PUSh's design, which assumes complete knowledge of object sharing patterns through manual annotations, KARD's algorithm is built for progressive identification of sharing patterns, which allows automated data race detection. Second, KARD implements efficient shared object identification, which is more than an order of magnitude faster than prior implemented identification strategies like Eraser's [49]. In contrast, PUSh relies on extensive manual effort, such as 35 hours to annotate and modify the *streamcluster* application, highly undesirable in testing settings [34]. Lastly, KARD implements protection interleaving to mitigate false positives and precisely identify the actual byte offsets accessed by conflicting threads. In contrast, to detect sub-object data races, PUSh page-aligns each structure member and array element, resulting in potential compatibility issues with external libraries [40] and substantial memory overhead.

Other hardware approaches. TxRace [62] uses Transactional Synchronization Extensions (TSX) to reduce the overhead of TSan. However, TSX suffers from false positives because any interrupt or cache line eviction can affect its behavior and its monitoring capacity depends on the cache size [33, 54]. REPT [15] uses Intel Processor Trace and crash dump to realize reverse debugging. However, REPT cannot detect transient data race bugs that do not produce a crash.

Static data race analysis. Static detectors [11, 31] can detect some data races during program compilation with no performance overhead. A major limitation of such approaches is that they can potentially report many false positives that developers want to avoid [34]. In particular, static techniques rely on imprecise pointer analysis [30, 55] that fails to accurately resolve function pointers [4] and cannot match data accesses to dynamically-allocated objects (e.g., heap-based). In contrast, dynamic detectors, including KARD, do not suffer from such problems, so they report fewer false positive data races than static analysis techniques.

10 CONCLUSION

KARD is a dynamic data race detector with a very low performance overhead of 7.0% with PARSEC and SPLASH-2x, and 5.3% with real-world applications, under the common testing scenario of 4 threads. KARD uses per-thread memory protection from Intel MPK to detect all data races involving inconsistent lock usages with a

key-enforced race detection algorithm. KARD overcomes various hardware limitations of MPK to have a low false positive rate and a modest memory overhead.

ACKNOWLEDGMENTS

We thank our shepherd, Brandon Lucia, and the anonymous reviewers for their valuable comments and suggestions. This work was partly supported by an Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone).

REFERENCES

[1] [n. d.]. https://openbenchmarking.org/system/1909082-HV-ICELAKETE36/TW20190905/cpuinfo.
[2] [n. d.]. sloccount(1) - Linux man page. https://linux.die.net/man/1/sloccount.
[3] Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. Transactional Memory with Strong Atomicity Using Off-the-shelf Memory Protection Hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. San Francisco, CA.
[4] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Vancouver, BC.
[5] Mark Adler. [n. d.]. pigz - Parallel gzip. https://zlib.net/pigz/.
[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2020. Data Center TCP (DCTCP). In *Proceedings of the 2020 ACM Special Interest Group on Data Communication (SIGCOMM)*. New Delhi, India.
[7] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. 2015. iThreads: A Threading Library for Parallel Incremental Computation. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey.
[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Toronto, ON.
[9] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-Only Region Conflict Exceptions. In *Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA.
[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, CA.
[11] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. In *Proceedings of 2018 ACM Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. Boston, MA.
[12] Stephen Blair-Chappell and Andrew Stokes. 2012. *Parallel Programming with Intel Parallel Studio XE*. John Wiley & Sons.
[13] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. Pacer: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, ON.
[14] Jonathan Corbet. 2015. Memory protection keys. https://lwn.net/Articles/643797/.
[15] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA.
[16] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, BC, Canada.
[17] Dormando. [n. d.]. memcached - a distributed memory object caching system. https://memcached.org.
[18] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J Boehm. 2012. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. Tucson, AZ.
[19] EnderUNIX Software Development Team. [n. d.]. EnderUNIX Aget: Multithreaded HTTP Download Accelerator. http://www.enderunix.org/aget/.
[20] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada.
[21] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-Threaded Applications. In *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys)*. Salzburg, Austria.
[22] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *Proceedings of 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Chicago, IL.
[23] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.
[24] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. Belgrade, Serbia.
[25] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. 2018. Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[26] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plain Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA.
[27] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.
[28] Intel. 2020. Inconsistent Lock Use. https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/reference/dependencies-problem-and-message-types/inconsistent-lock-use.html.
[29] Ayal Itzkovitz and Assaf Schuster. 1999. MultiView and Millipage—Fine-Grain Sharing in Page-Based DSM. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. New Orleans, LA.
[30] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[31] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and Accurate Static Data-Race Detection for Concurrent Programs. In *Proceedings of the International Conference on Computer Aided Verification*. Berlin, Heidelberg.
[32] Chris Lattner and Vikram Adve. 2004. 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)*.
[33] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE)*. Chicago, IL.
[34] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. 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)*. Huntsville, ON.
[35] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. 2016. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.
[36] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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.
[37] Brandon Lucia and Luis Ceze. 2009. Finding Concurrency Bugs with Context-Aware Communication Graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY.
[38] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. 2010. 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, CA.
[39] David Mulnix. 2019. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.
[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland.
[41] NGINX Inc. [n. d.]. NGINX High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com.
[42] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. 2012. 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.

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

[44] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Phoenix, AZ.

[45] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340.

[46] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. 2009. ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs. In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Washington, DC.

[47] M. Rajashekhar and C. Janet. [n. d.]. twemperf - A tool for measuring memcached server performance. https://zlib.net/pigz/.

[48] Caitlin Sadowski and Jaeheon Yi. 2014. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*.

[49] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[50] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event, USA.

[51] selenic. [n. d.]. smem memory reporting tool. https://www.selenic.com/smem/.

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

[53] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. 2007. 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.

[54] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[55] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*.

[56] The Apache Software Foundation. [n. d.]. ab - Apache HTTP server benchmark tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[57] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammer, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[58] Dmitry Vyukov. [n. d.]. ThreadSanitizerFoundBugs. https://github.com/google/sanitizers/wiki/ThreadSanitizerFoundBugs.

[59] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful.. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada.

[60] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Valencia, Spain.

[61] Jie Yu and Satish Narayanasamy. 2009. A Case for an Interleaving Constrained Shared-memory Multi-processor. In *Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Austin, TX, USA.

[62] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2016. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[63] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2017. 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.

[64] Diyu Zhou and Yuval Tamir. 2019. 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.

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