FINDING RACE CONDITIONS IN KERNELS: FROM FUZZING TO SYMBOLIC EXECUTION

A Dissertation
Presented to
The Academic Faculty

By

Meng Xu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2020

FINDING RACE CONDITIONS IN KERNELS: FROM FUZZING TO SYMBOLIC EXECUTION

Date Approved: July 16, 2020

To my family,

for the unconditional support.

ACKNOWLEDGEMENTS

My Ph.D. thesis would not be possible without the help from a number of people.

First and foremost, I would like to express my sincere gratitude and appreciation to my advisor, Prof. Taesoo Kim, who not only inspires me to pursue the research direction in system and software security, but also provides valuable guidance and generous support that allows me to succeed in this direction. His innovative ideas and comments helped me shape almost all my research projects in the six years. But more importantly, through the advice, he taught me with arguably the most essential skill for an independent researcher: critical thinking. I am forever thankful to the training Taesoo provides me and I hope that one day I can be a good advisor to my future students as Taesoo has been to me.

I would like to express my special thanks to my collaborators who also mentored me both at Georgia Tech and during my internships. In particular, many thanks to Prof. Wenke Lee for the thought-provoking discussions and invaluable career advice. And my sincere gratitude goes to Prof. Michael Backes, Dr. Sam Blackshear, and Dr. Marcus Peinado for kindly hosting me at CISPA, Facebook, and Microsoft Research, respectively. Many parts of the thesis are derived from our fruitful collaboration and those research projects will not be successful without their trust, support, and constructive comments. I would also like to thank Prof. Alessandro Orso and Prof. Brendan D. Saltaformaggio for taking time to serve on my thesis committee. Their insightful feedback has improved this thesis significantly.

This thesis is a product of great team work. I have been fortunate to collaborate with many brilliant and friendly colleagues in SSLab, IISP, or wherever I interned at. They not only help me with research challenges but also make everyday life enjoyable: I would like to take this opportunity to thank: Prof. Xinyu Xing, Prof. Byoungyoung Lee, Prof. Chengyu Song, Prof. Kangjie Lu, Prof. Yeongjin Jang, Dr. Sangho Lee, Dr. Yang Ji, Dr. Hong Hu, Dr. ChulWon Kang, Dr. Daehee Jang, Dr. Paul England, Dr. Manuel Huber, Dr. Ruian Duan, Dr. Ming-Wei Shih, Sanidhya Kashap, Insu Yun, Chenxiong Qian, Seulbae Kim, Ren

Ding, Fan Sang, Hanqing Zhao, Wen Xu, Zhichuang Sun, Jinho Jung, and Jungyeon Yoon. In addition, I am also grateful to our support staff, Elizabeth Ndongi, Sue Jean Chae, and Trinh Doan for their help in all the administrative work.

Last but not least, to my parents back in China, I always thank you for your unconditional love, patience, and support throughout my years at Georgia Tech. I am forever indebted to Pei Wang for her love, patience, support, and encouragement, without which I could not have survived the most difficult times in my PhD journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The scale and pervasiveness of concurrent software pose challenges for security researchers: race conditions are more prevalent than ever, and the growing software complexity keeps exacerbating the situation — expanding the arms race between security practitioners and attackers beyond memory errors. As a consequence, we need a new generation of bug hunting tools that not only scale well with increasingly larger codebases but also catch up with the growing importance of race conditions.

In this thesis, two complementary race detection frameworks for OS kernels are presented: multi-dimensional fuzz testing and symbolic checking. Fuzz testing turns bug finding into a probabilistic search, but current practices restrict themselves to one dimension only (sequential executions). This thesis illustrates how to explore the concurrency dimension and extend the bug scope beyond memory errors to the broad spectrum of concurrency bugs. On the other hand, conventional symbolic executors face challenges when applied to OS kernels, such as path explosions due to branching and loops. They also lack a systematic way of modeling and tracking constraints in the concurrency dimension (*e.g.*, to enforce a particular schedule for thread interleavings) The gap can be partially filled with novel techniques for symbolic execution in this thesis.

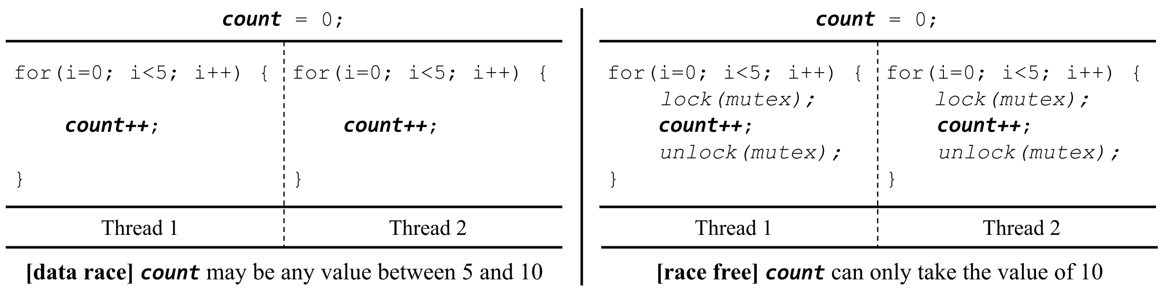
CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Designing and maintaining operating system (OS) kernels are not easy. With the constant development for performance optimizations and new features, popular kernel software have grown too large to be bug-free. For example, the two most popular file systems, `ext4` [1] and `btrfs` [2], with 50K and 130K lines of code, respectively, witnessed 54 [3] and 113 [4] bugs reported in 2018 alone. The Linux kernel version 4.15 (deployed by default in Ubuntu 18.04) witnessed over 650 bugs reported in a few months after its debut [5]. A bug in kernel can wreak havoc on the user, as it not only results in reboots, deadlock, or corruption of the whole system [6], but also poses severe security threats [7, 8, 9], such as privilege escalation [10, 11], information leaks [12], and denial of service [13]. Thus, finding and fixing bugs is a constant yet essential activity during the entire life cycle of any OS kernel.

Furthermore, the aggressive adoption of concurrent programming in kernel software makes bug hunting even more complicated. In the current multi-core era, concurrency has been a major thrust for performance improvements, especially for system software. As is evident in kernel and file system evolution [6, 14, 15, 16], a whole zoo of programming paradigms is introduced to exploit multi-core computation, including but not limited to asynchronous work queues, read-copy-update (RCU), and optimistic locking such as sequence locks. However, alongside performance improvements, concurrency bugs also find their ways to the code base and have become particularly detrimental to the reliability and security of file systems due to their devastating effects such as deadlocks, kernel panics, data inconsistencies, and privilege escalations [9, 17, 18, 7, 19, 4, 20, 3].

In the broad spectrum of concurrency bugs, race conditions are an important class in



```
count = 0;
for(i=0; i<5; i++) {        for(i=0; i<5; i++) {

    count++;                    count++;

}                           }
       Thread 1                    Thread 2
```

[data race] ***count*** may be any value between 5 and 10

```
count = 0;
for(i=0; i<5; i++) {        for(i=0; i<5; i++) {
    lock(mutex);                lock(mutex);
    count++;                    count++;
    unlock(mutex);              unlock(mutex);
}                           }
       Thread 1                    Thread 2
```

[race free] ***count*** can only take the value of 10

Figure 1.1: A simple illustration on the concept of data races. In the example on the left, accesses to variable `count` is raced in the two threads. As a consequence, the final value of `count` can be anything between 5 and 10. To fix this issue, a typical approach is to place mutually exclusive locks around the accesses to `count` to ensure that there should be at the maximum one thread mutating the `count` variable—as shown in the example on the right.

which two threads erroneously access a shared memory location without proper synchronization or ordering, as showcased in Figure 1.1. Obstructed by the non-determinism in thread interleavings, race conditions are notoriously difficult to detect and diagnose, as they only show up in rare interleavings that require precise timing to trigger. Even worse, unlike memory errors that tend to crash the system immediately upon triggering, race conditions do not usually raise visible signals in the short term and are often identified retrospectively when analyzing assertion failures or warnings in production logs [21].

Although OS kernels have been increasingly hardened throughout the years, *e.g.*, kASLR [22], kCFI [23, 24], and UniSan [25], these defenses are effective against memory errors only (*e.g.*, stack and buffer overflows) and have limited success in taming attacks that exploit concurrency bugs. As the state of the practice, kernel developers often rely on stress testing to find race conditions proactively [26, 27]. By saturating a kernel component (*e.g.*, a file system) with intensive workloads, the chance of triggering uncommon thread interleavings, and thus race conditions, can be increased. However, while useful, stress testing has significant shortcomings: handwritten test suites are far from sufficient to cover the enormous state space in kernel execution, not to mention keeping up with the rapid increase in code size and complexity. For example, we found a data race in `btrfs` (Figure 2.1) that could render the management of the reserve space ineffective and might eventually cause integer overflows. Both `xfstests` and `fsck`—the most commonly used file system testing suites—miss this case. In fact, none of the test cases in `xfstests` or `fsck` even attempt to stress the reserve management part of `btrfs`, which is completely internal and implementation-dependent.

The incompleteness of hand-crafted test suites highlights the urgent need for an automated approach to explore execution states in OS kernels, which are essentially large and complex software system with decades of development effort accumulated. Fortunately, recent years have witnessed a surge of research in this direction, from expert communities not only in the security domain, but also in systems, software engineering, and programming languages. Prominent works include (but not limited to) KINT [28], SymDrive [29], DrChecker [30], kAFL [31], Syzkaller [32], Janus [9] and its follow-up, Hydra [17]. These works can be broadly classified into two categories:

- *Fuzz testing*, (*a.k.a*, fuzzing), executes a program at (almost) *native speed* with *concrete inputs* for each execution. A typical fuzzer explores a target program by tweaking inputs using simple rules (*e.g.*, bit flips). Coupled with evolutionary algorithms (*a.k.a*, genetic programming), the fuzzer will eventually be able to produce complicated test cases that are hard to be even contemplated by a human expert, which may expose corner and buggy states in program execution such as crashing. As a result, fuzzing favors depth of execution over completeness of checking, *i.e.*, fuzzing is usually very effective in finding bugs deep in the program but do not provide any bug-free guarantee on of states already checked.

- *Symbolic execution*, which sometimes might also be referred to as model checking, is almost the complete opposite of fuzzing. Symbolic execution attempts to enumerate possible program behaviors by *statically emulating* the program, either at source code or intermediate representation (IR) level, with *symbolic* inputs that do not have concrete values (*e.g.*, a symbolic integer x vs a concrete value `10`). During the emulation, a symbolic executor typically collects a set of constraints for the symbolic input (*e.g.*, $x < 0$) in order for the execution to reach a given state (*e.g.*, a buffer overflow site) and dispatch the constraints to a satisfiability modulo theories (SMT) solver for satisfiability checking. As a result, symbolic execution often favors completeness over depth of examination as the deeper it explores, the more constraints it collects, which might eventually exceed the capability of SMT solvers.

However, a vast majority these works have focused on memory safety violations only (*e.g.*, integer overflows, buffer overread, use-after-free), with less attention on concurrency bugs, especially race conditions. More importantly, these works aim at exploring the *sequential* aspect of program execution only and fail to treat *concurrency* as a first-class citizen. This thesis is a partial effort to fill this gap by bringing both fuzz testing and symbolic execution to the concurrency dimension with a focus on race condition detection. The blueprint is outlined in this chapter and the complete framework is presented in the subsequent chapters in this thesis. The rest of this chapter will 1) briefly explain the challenges in applying fuzzing and symbolic execution to highly concurrent programs; 2) highlight the promising directions in solving these challenges; and 3) summarize the contribution in this thesis work: an automated race condition detection framework.

1.2 Navigating the Concurrency Dimension via Fuzz Testing

In recent years, coverage-guided fuzzing has proven a useful technique in testing large and complex software systems, with thousands of vulnerabilities found in userspace programs [33, 34, 35, 36, 37]. Without a doubt, OS kernels can be fuzzed, and generic OS fuzzers [32, 31, 38] have demonstrated their viability with over 200 bugs found. In addition, file system-specific fuzzers, Janus [9] and Hydra [17], have extended the scope of kernel fuzzing from memory errors into a broad set of semantic bugs, while the data race-specific fuzzer, Razzer [39], has shed lights on data race detection by combining fuzzing and static analysis. At the core of these fuzzers is the coverage measurement scheme, which summarizes unique program behaviors triggered by a given input in bitmaps. The fuzzer compares per-input coverage against the accumulated coverage bitmaps to measure the "novelty" of the input and determines whether it should serve as the seed for future fuzzing rounds.

However, to re-iterate this point, almost all existing coverage-guided fuzzers focus on tracking the *sequential* aspect of program execution only and fail to treat *concurrency* as a first-class citizen. To illustrate, branch coverage (*i.e.*, control flow transition between basic blocks) has been the predominant coverage measurement metric. But such a metric captures little information about thread interleavings: different interleavings are likely to result in the same branch coverage (as shown in Figure 3.1), while only a small fraction may trigger a data race (as shown in Figure 3.2).

With the sequential view of program execution, existing kernel fuzzers have been very effective in mutating and synthesizing single-threaded syscall sequences based on seed traces [40, 41] to maximize branch coverage. But no heuristics have been proposed in synthesizing multi-threaded sequences to maximize thread interleaving coverage. This applies to Razzer [39] as well since its fuzzing component is used to generate single-threaded syscall traces only instead of multi-threaded traces. Furthermore, given that data races often lead to silent failures, treating kernel panics or assertion failures as the only types of bug signals is not sufficient: a data race checker that understands the semantics in kernel synchronization primitives and is capable of pinpointing data races in a dynamic execution trace is needed as the signal raiser.

To bring coverage-guided fuzzing to the concurrency dimension, in this thesis, we present KRACE, an end-to-end fuzzing framework that fills the gap with new components in three fundamental aspects in kernel fuzzing:

Coverage tracking [section 3.1] KRACE adopts two coverage tracking mechanisms. Branch coverage is tracked as usual to capture code exploration in the sequential dimension, analogous to the line coverage metric used in unit testing. In addition, to approximate the exploration progress in the concurrency domain, KRACE proposes a novel coverage metric: alias instruction pair coverage, short for alias coverage. Conceptually, if we could collect all pairs of memory access instructions X↔Y such that X in one thread *may-interleave* against Y in another thread, alias coverage tracks how many such interleaving points have been covered in execution. Consequently, if the growth of alias coverage stalls, it signals the fuzzer to stop probing for new interleavings in the current multi-threaded seed input.

Input generation [section 3.2] KRACE generates and mutates individual syscalls according to a specification [42, 32], which is not new. The novel part of KRACE lies in evolving multi-threaded seeds and merging them in an interleaved manner to preserve already-found coverage as well as to maximize the chances of inducing new interleavings. Another job of the input generator is to produce thread scheduling, in other words, to explore the hidden input space in concurrent programs. Although enforcing fine-grained control over thread scheduling is possible [18], the scheduling algorithm does not scale to whole-kernel concurrency, as the latter consists of not only user threads, but also background threads internally forked by file systems, work queues, the block layer, loop devices, RCUs, etc., and the total number of contexts often exceeds 60 at runtime. As a result, KRACE adopts a lightweight delay injection scheme and relies on the alias coverage metric as feedback to determine whether more delay schedules are needed.

Bug manifestation [section 3.3] KRACE incorporates an in-house developed detector to reason about data races given an execution trace. In essence, KRACE hooks every memory access and for each pair of accesses to the same memory address, KRACE checks whether

- they belong to two threads and at least one is a memory write;
- these two accesses are strictly ordered (*i.e.*, happens-before relation); and
- at least one shared lock exists that guards such accesses (*i.e.*, lockset analysis).

The challenges for KRACE lie in modeling the diverse set of kernel synchronization mechanisms comprehensively, especially those uncommon primitives such as optimistic locking, RCU, and ad-hoc schemes implemented in each file system.

KRACE adopts the software rejuvenation strategy to avoid the aging OS problem, *i.e.*, every execution is a fresh run from a clean-slate kernel and empty file system image.

Doing so trades performance for trackability and debuggability but is worthwhile for data race detection. The reason is that state exploration will gradually catches up and bypasses conventional speed-oriented fuzzers (*e.g.*, Syzkaller) upon saturation. KRACE also decouples data race checking from state exploration. Unlike prior works where the bug checker runs inline in each execution, in KRACE, the checker only kicks in when new coverage (either branch or alias) is reported. This prevents the expensive data race checking from slowing down the state exploration while still preserving the opportunity to test every new execution state found through fuzzing. The checking progress will eventually catch up when the coverage growth is toward saturation. More details of KRACE will be presented in chapter 3.

We evaluated KRACE by fuzzing two popular and heavily tested kernel file systems (`ext4` and `btrfs`) in recent kernel versions and we found 23 data races, nine of which are confirmed as potentially harmful races, and 11 are benign races (for performance or allowed by the POSIX specification).

1.3 Finding Double-fetch Bugs with Symbolic Model Checking

We use *double-fetch bugs* as a case study and entry point for applying symbolic checking in the concurrency dimension. The Bochspwn project [43] first introduced *double-fetch bugs* in the context of the Windows kernel while Wang *et al.* further studied *double-fetch bugs* in the Linux kernel [44]. A *double-fetch bug* is a special type of race condition bug in which (typically during syscall execution) the kernel reads a particular userspace memory region more than once with the assumption that the content in the accessed region does not change across reads. However, this assumption is not valid. A concurrently running user thread can "scramble" the same memory region in between kernel reads, leading to data inconsistencies in the execution path, which can lead to exploitable vulnerabilities such as sanity check bypassing, buffer overflow, and confused deputy. In reality, researchers have exploited *double-fetch bugs* to escalate privileges on Windows OS [45, 46].

What makes *double-fetch bug* detection an important problem is that, in kernel, it is common to intentionally read data multiple times from the userspace for performance reasons. We call this situation a *multi-read*. To illustrate, consider fetching a variable-length message with a potentially maximum size of 4 KB from the userspace. One approach is to always pre-allocate a 4 KB buffer and copy 4 KB from the userspace in one shot. However, in most cases, this wastes memory and CPU cycles if the effective message payload is 64 bytes or less. Hence, the kernel handles this scenario by first fetching a 4-byte `size` variable and later allocating the buffer and fetching the `size`-byte message. A quick scan over the Linux kernel reveals that there are over 1,000 *multi-reads*. Then, a follow-up question would be: How many of them are real *double-fetch bugs*? Until now, the only way to answer this question was to manually vet the complicated source code of all *multi-reads*. However, this is certainly a scale beyond manual vetting. It therefore becomes a pressing problem that we have to both 1) formally define and distinguish *double-fetch bugs* and *multi-reads* and 2) automatically verify each *multi-read* to check whether it is a bug.

Unfortunately, neither aspect has been addressed perfectly in prior works. Bochspwn [43] defines *multi-reads* as at least two memory reads from the same userspace address within a short time frame, while Wang *et al*. [44] defines *multi-reads* based on a few empirical static code patterns. Due to the imprecise definitions, both works result in many false positives (i.e., incorrectly identified bugs) and false negatives (i.e., missing bugs). More importantly, neither of them can systematically distinguish *double-fetch bugs* from *multi-reads* in definition and they completely leave it to manual verification.

In this thesis, we present DEADLINE, an automatic tool to statically detect *multi-reads* and *double-fetch bugs* with both high precision and code coverage. In particular, DEADLINE covers all drivers, file systems, and other peripheral modules that can be compiled under the x86 architecture for both Linux and the FreeBSD kernels.

To guide DEADLINE to detect *double-fetch bugs*, we first formally model and mathematically distinguish *double-fetch bugs* from *multi-reads*. In essence, a *multi-read* becomes a *double-fetch bug* when

- two fetches are guaranteed to read from an overlapped userspace memory region,
- a relation between the two fetches is established based on the values in the overlap,
- the relation can be destroyed by a race condition that changes the value in the overlap.

With these definitions, DEADLINE detects *double-fetch bugs* in two steps. In the first step, DEADLINE tries to find as many *multi-reads* as possible and also builds execution paths for each *multi-read* by compiling the kernel source to LLVM intermediate representation (IR) followed by a static code analysis. In the second step, DEADLINE follows the execution paths to vet whether a *multi-read* turns into a *double-fetch bug*. To do this, DEADLINE first transforms the LLVM IR into a symbolic representation (SR) in which each variable is represented by a symbolic expression. After this procedure, DEADLINE detects a *double-fetch bug* by solving symbolic constraints on the SR in accordance with the *double-fetch bug* definitions. A satisfiable result indicates that a *double-fetch bug* exists, while an unsatisfiable result means a bug does not exist.

Although the process sounds intuitive, applying it to kernel code imposes several practical challenges. For example, to detect *multi-reads*, DEADLINE needs to systematically explore paths to collect *multi-reads*, and further trim irrelevant instructions and linearize these execution paths. For *double-fetch bug* vetting, DEADLINE needs to symbolize memory reads and writes, and emulate common library functions. DEADLINE embodies various techniques to address these challenges. In particular, instead of using empirical lexical matching [44], it relies on program analysis to collect *multi-reads* and further applies backward slicing and loop unrolling to prune the execution path. For symbolic checking, we propose our own memory model in extension to the model used by traditional symbolic executors [47, 48, 49] to encode access sequence and memory object information. We also write manual symbolic rules to emulate library functions, which alleviate DEADLINE from having to handle the intricacies in these functions.

With DEADLINE, we find and report 23 new bugs in the Linux kernel and a new bug in the FreeBSD kernel. Besides detection, we complete the analysis cycle of *double-fetch* *bugs* by discussing how to exploit *double-fetch bugs* as well as four generic ways to fix *double-fetch bugs* based on our experience in patching these bugs.

1.4 Thesis Contribution

In summary, this thesis makes the following contributions to advance the research on finding race conditions in OS kernels:

Concept :

- Two novel concepts are proposed in KRACE to bring coverage-guided fuzzing to highly concurrent programs: alias coverage and interleaved multi-threaded syscall sequence merging. These concepts serve as the first step toward adapting fuzzing for a wide range of concurrency bugs beyond data races.

- In DEADLINE, we proposed a formal and precise definition of *double-fetch bugs* and such a formal definition helps to eliminate the need to manually verify whether a *multi-read* is a *double-fetch bug*, and hence, significantly reduces the manual effort in detecting this special type of TOCTOU bugs.

Design and implementation :

- KRACE's data race checker encodes a comprehensive model of kernel synchronization mechanisms in the form of over 100 kernel patches (for code instrumentation), which are regularly updated as the kernel upgrades.

- DEADLINE sheds lights on the design and implementation of an end-to-end system that automatically vet kernel code with a tailored symbolic execution model specifically designed for *double-fetch bug* detection.

Broader impact :

- KRACE has found 23 data races and will be continuously running to find new cases. We will open-source KRACE as well as the collection of syscall primitives for multi-threaded execution as quality seeds for future concurrent file system fuzzing research.
- With DEADLINE, we found and reported 23 new bugs in the Linux kernel and a new bug in the FreeBSD kernel. We further proposed four generic strategies to patch and prevent *double-fetch bugs* in future kernel development based on our study and the discussion with kernel maintainers.

CHAPTER 2
BACKGROUND

The past three decades have witnessed several efforts to find race conditions using various techniques. In this chapter, we show a few examples on race conditions of various flavors, discuss the types of approaches that prior works have taken, and introduce both coverage-guided fuzzing and symbolic model checking as generic bug finding techniques.

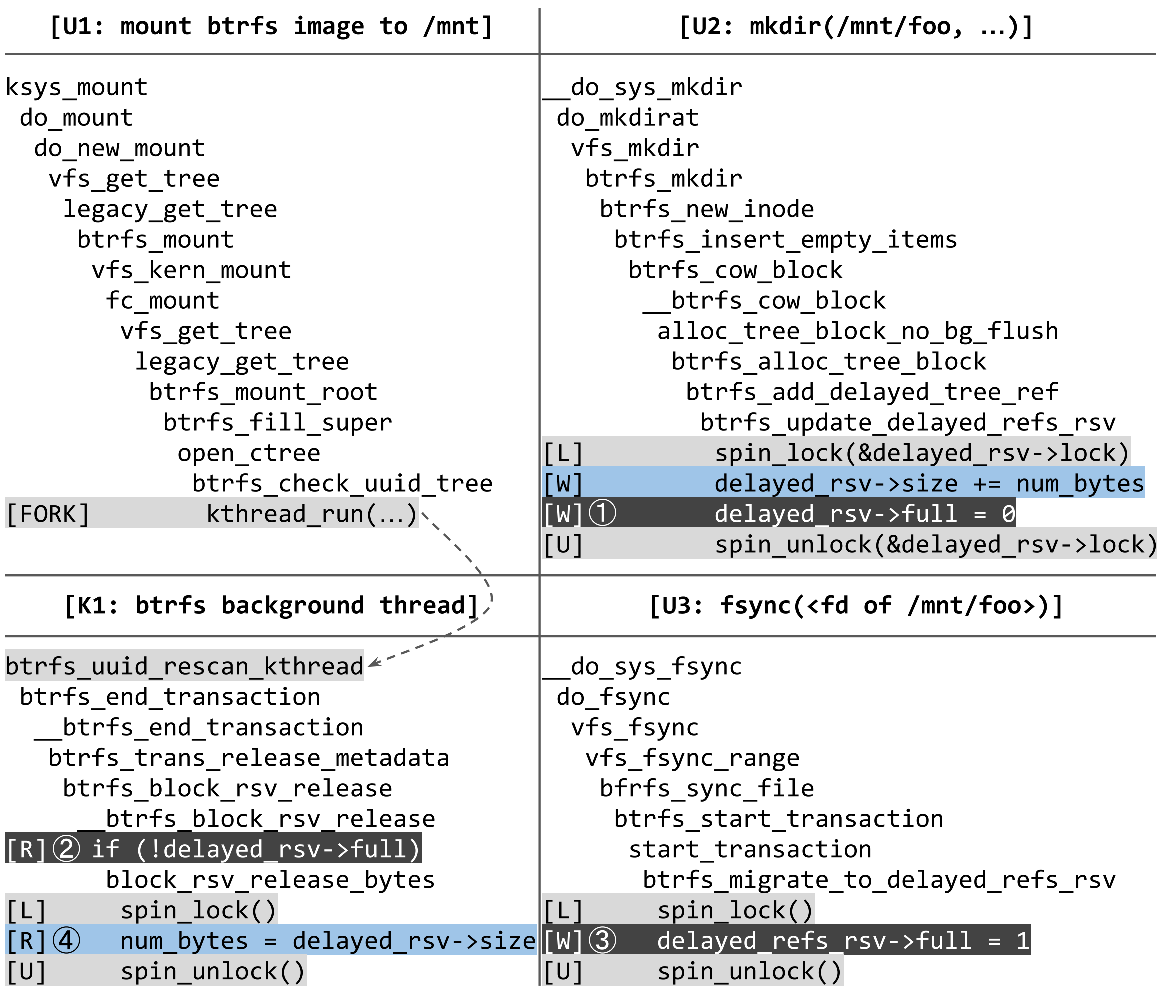
2.1 Race Condition Examples

2.1.1 Data races

Intuitively, a data race is caused by two threads trying to perform unordered and unprotected memory operations to the same address. Figure 2.1 shows two data races found by KRACE that happen to make a complete scenario. The read of `full` is in race with both writes, as the read is not protected by the corresponding `delayed_rsv->lock` as is done on the writers' side. According to `btrfs` developers, this results in ineffective management of the reserve space internally used by `btrfs`, in particular, delays in releasing the reserved space or space releasing followed by reservation instead of migration from one reserve to another. Reflected in the call stack, if the execution takes the order of ①→②→③→④, then `block_rsv_release_bytes` is inadvertently releasing bytes that will be used by the `fsync`. Such a case might eventually cause integer overflows in the reserve space but would probably require thousands of concurrent file operations to trigger.

In summary, data race is a special type of race condition, and hunting data races in complex software involves two facets:

- how to confirm an execution is racy, and
- how to produce meaningful executions by exploring both code and thread-scheduling.



```
[U1: mount btrfs image to /mnt]

ksys_mount
 do_mount
  do_new_mount
   vfs_get_tree
    legacy_get_tree
     btrfs_mount
      vfs_kern_mount
       fc_mount
        vfs_get_tree
         legacy_get_tree
          btrfs_mount_root
           btrfs_fill_super
            open_ctree
             btrfs_check_uuid_tree
[FORK]        kthread_run(...)
```

```
[U2: mkdir(/mnt/foo, ...)]

__do_sys_mkdir
 do_mkdirat
  vfs_mkdir
   btrfs_mkdir
    btrfs_new_inode
     btrfs_insert_empty_items
      btrfs_cow_block
       __btrfs_cow_block
        alloc_tree_block_no_bg_flush
         btrfs_alloc_tree_block
          btrfs_add_delayed_tree_ref
           btrfs_update_delayed_refs_rsv
[L]          spin_lock(&delayed_rsv->lock)
[W]          delayed_rsv->size += num_bytes
[W]①         delayed_rsv->full = 0
[U]          spin_unlock(&delayed_rsv->lock)
```

```
[K1: btrfs background thread]

btrfs_uuid_rescan_kthread
 btrfs_end_transaction
  __btrfs_end_transaction
   btrfs_trans_release_metadata
    btrfs_block_rsv_release
     __btrfs_block_rsv_release
[R]② if (!delayed_rsv->full)
      block_rsv_release_bytes
[L]    spin_lock()
[R]④   num_bytes = delayed_rsv->size
[U]    spin_unlock()
```

```
[U3: fsync(<fd of /mnt/foo>)]

__do_sys_fsync
 do_fsync
  vfs_fsync
   vfs_fsync_range
    bfrfs_sync_file
     btrfs_start_transaction
      start_transaction
       btrfs_migrate_to_delayed_refs_rsv
[L]     spin_lock()
[W]③    delayed_refs_rsv->full = 1
[U]     spin_unlock()
```

Figure 2.1: A data race found by KRACE. This figure shows the complete call stack, thread ordering information, and locking information when the data race happens and the inconsistency it may cause (①-④). In particular, if the execution happens in the order of ①②③④, the `uuid_rescan` thread will think the reserve is not full and release the bytes added, which is not intended as the `fsync` will be using them later.

2.1.2 Time-to-check vs time-to-use bugs

Racing to access a single memory location not only leads to inconsistencies among the two threads (*i.e.*, a data race) but also confusion in other threads which may not be aware that the underlying memory might change (in other words, the operation is not atomic) during its processing, leading to bugs named time-to-check vs time-to-use errors, short for TICTOU bugs. A typical example is *double-fetch bugs* in kernels.

In modern operating systems, virtual memory is divided into userspace and kernel-space regions. Most notably, the userspace region is separated for each process running in the system, creating an illusion of exclusive address space for each program. Userspace memory can be accessed from all threads running in that address space as well as from kernel. On the other hand, the kernel memory is system-wide and is accessible from the kernel only.

Furthermore, although userspace memory is accessible to the kernel, in practice, the kernel almost never directly dereferences an address supplied by user processes, as any corrupted address, be it by mistake or by intention, will crash the whole system. Instead, if the kernel requires userspace data for execution (as in the case of many driver IOCTL routines), it first duplicates the data into kernel memory and then works on its internal copy. Special schemes, termed *transfer functions*, are provided for this purpose, such as `copy_from_user`, `get_user` in Linux, and `copyin`, `fuword` in FreeBSD. These schemes not only perform data transfer, but also actively validate userspace accesses and handle illegal addresses or page faults. In fact, extensive manual instrumentation (e.g., the `__user` mark) are placed to ensure that userspace memory can be accessed only through transfer functions.

Given the limited number of arguments a user process can directly pass to the kernel for a syscall (e.g., maximum six arguments on x86_64), pointers pointing to block structures in userspace memory are often passed to handle large or complex requests. In this case, the kernel often needs to refer back to userspace memory during the syscall. Theoretically, any *multi-read* can be re-designed to a single-read by pre-defining the shape of the buffer (e.g., the maximum size) and always copying the whole buffer in one shot. However, in practice, this pattern is rarely used due to the waste of memory and CPU cycles, especially when the effective payload is often much smaller than the maximum allowed. Instead, what is typically done in kernel is to first fetch a request header, often a few bytes only, and then construct the whole request based on the information in the header. Wang *et al*. [44] identified three scenarios of this pattern, namely, *size checking*, where the actual length of the request depends on a `size` variable; *type selection*, where the actual length of the request

```c
void mptctl_simplified(unsigned long arg) {
  mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
  MPT_ADAPTER *iocp = NULL;

  // first fetch
  if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
    return -EFAULT;

  // dependency lookup
  if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
    return -EFAULT;

  // dependency usage
  mutex_lock(&iocp->ioctl_cmds.mutex);
  struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;

  // second fetch
  if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
    return -EFAULT;

  // BUG: kfwdl.iocnum might not equal to khdr.iocnum
  mptctl_do_fw_download(kfwdl.iocnum, ......);
  mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

Figure 2.2: A dependency lookup *double-fetch bug*, adapted from `__mptctl_ioctl` in file `drivers/message/fusion/mptctl.c`

depends on the `opcode` of the action performed; and *shallow copy*, where the request header contains a pointer to the second buffer in userspace.

Our analysis confirms these common scenarios but also discovers more interesting reasons and patterns for *multi-reads*.

Dependency lookup. As shown in Figure 2.2, in the case where there could be multiple handlers for a request, a lookup, based on the request header, is first performed to find the intended handler, and later the whole request is copied in.

Protocol/signature checking. As shown in Figure 2.3, the request header is first checked against a pre-defined protocol number. The kernel rejects the request early if the protocol is not honored.

Information guessing. As shown in Figure 2.4, when certain information is missing, the kernel might first guess this piece of information via a sequence of selective reads from the userspace and later fetch in the whole data. A common rationale behind these cases is to abort the processing early if the request is erroneous and save the cost of buffer allocation and a full request copying.

```
void tls_setsockopt_simplified(char __user *arg) {
  struct tls_crypto_info header, *full = /* allocated before */;

  // first fetch
  if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
    return -EFAULT;

  // protocol check
  if (header.version != TLS_1_2_VERSION)
    return -ENOTSUPP;

  // second fetch
  if (copy_from_user(full, arg,
        sizeof(struct tls12_crypto_info_aes_gcm_128)))
    return -EFAULT;

  // BUG: full->version might not be TLS_1_2_VERSION
  do_sth_with(full);
}
```

Figure 2.3: A protocol checking *double-fetch bug*, adapted from `do_tls_setsockopt_txZ` in file `net/tls/tls_main.c`

```
void con_font_set_simplified(struct console_font_op *op) {
  struct console_font font;

  if (!op->height) { /* Need to guess font height [compat] */
    u8 tmp, __user *charmap = op->data;
    int h, i;
    for (h = 32; h > 0; h--)
      for (i = 0; i < op->charcount; i++) {
        // first batch of fetches
        if (get_user(tmp, &charmap[32*i+h-1]))
          return -EFAULT;
        if (tmp)
          goto nonzero;
      }
    return -EINVAL;
nonzero:
    op->height = h;
  }

  font.height = op->height;
  // second fetch
  font.data = memdup_user(op->data, size);
  if (IS_ERR(font.data))
    return -EINVAL;

  // BUG: the derived font.height might not match with font.data
  do_sth_with(&font);
}
```

Figure 2.4: An information guessing *double-fetch bug*, adapted from `con_font_set` in file `drivers/tty/vt/vt.c`

However, it is worth noting that the goal of this analysis and categorization is not to enumerate all possible patterns that might cause *double-fetch bugs*; instead, it shows the diversity of TICTOU bugs as well as the importance of finding a generic, formal, yet comprehensive definition of *multi-reads* and *double-fetch bugs* that can unify all these patterns as well as potentially undiscovered ones.

2.2 Hunting Security Vulnerabilities via Fuzz Testing

2.2.1 Fuzzing in general

Fuzzing has proven to be a practical approach to find bugs in today's software stack, both in the userspace [33, 50, 37, 51, 52, 53, 54] and in the kernel space [31, 9, 17, 32, 55, 42]. Unfortunately, existing works cannot be trivially adopted for data race fuzzing. One reason is that the main focus of fuzzing has been on finding memory corruptions or triggering assertions. Although Hydra [17] extends the scope beyond memory errors into semantic bugs in file systems, it does not provide any insight into finding data races.

Moreover, since modern coverage-guided fuzzing originates and prospers from testing single-threaded programs such as binutils, encoder/decoders, and the CGC and LAVA-M fuzzing benchmarks, recent fuzzing efforts have focused on optimizing fuzzers' performance on single-threaded executions too, such as approximating sequential execution with neural networks [51]. Not surprisingly, when the fuzzing practice is carried down to the OS level [32, 31, 55, 56, 57, 58, 42, 59], the same sequential view of program execution is inherited.

Although generating structured inputs has been a challenge for kernel fuzzing, many improvements have been proposed. For example, MoonShine [40] captures dependencies between syscalls and DIFUZE [41] generates interface-aware inputs. However, lacking a coverage metric and a seed evolution algorithm to handle state exploration in the concurrency dimension, existing OS fuzzers miss the opportunities to find the broad spectrum of concurrency bugs, including data races. The motivation behind KRACE is to fill this gap and to bring coverage-guided fuzzing to the concurrency dimension.

2.2.2 Fuzzing for data races in OS kernels

Code/thread-schedule exploration. The effectiveness of a data race checker depends not only on the detection algorithm but also on how well the checker can explore execution states and cover as many code paths and thread interleavings as possible. For code path exploration, prior detectors mostly rely on manually written test suites [18, 60, 61] that do not capture complicated cases. As shown in Figure 2.1, triggering the data race would require a user thread to `mkdir` on the same block the background `uuid_rescan` thread is working on, which (almost) in no way can be specified in manually written test cases. An alternative is to enumerate code paths statically [62, 63, 64, 65, 66], but this is not scalable. Recent OS fuzzers adopt specification-based syscall synthesization [9, 17, 32, 42]. However, these fuzzers mostly focus on generating sequential programs instead of multi-threaded programs and are not intended to explore interleavings in syscall execution. KRACE adopts a similar synthesization approach, but instead of focusing on single-threaded sequences, KRACE evolves multi-threaded programs.

In the case of thread-schedule exploration, prior approaches fall into three categories, in decreasing order of scalability but increasing order of completeness: 1) stressing the random scheduler with multiple trials [26]; 2) injecting delays at runtime [60, 61, 21]; and 3) enumerating every possible thread interleaving [18, 39]. KRACE uses delay injection, a trade-off among scalability, practicality, and completeness.

Data race checking. Four prominent works [67, 18, 39, 68] lay the ground for data race checking in a fuzzy way:

DataCollider [67] is the first work that tackles this problem by using randomized sampling of a small number of memory accesses in conjunction with code breakpoint and data breakpoint facilities for efficient sampling. DataCollider is simple enough to detect several bugs in the Windows kernel modules. A similar strategy is used by Syzkaller [32] with its Kernel Concurrent Sanitizer [68] (KCSan) module.

KCSan is a dynamic data race detector that uses compiler instrumentation, *i.e.*, software watchpoints instead of hardware watchpoints, to detect bugs on non-atomic accesses that violate the Linux kernel memory model [69] using happens-before analysis.

SKI [18] focuses on comprehensive enumeration of thread schedules with the PCT algorithm [70] and hardware breakpoints. However, SKI permutes user threads only to find data races in the syscall handlers and thus forgoes the opportunities to find data races in kernel background threads. Furthermore, even with user threads only, the number of permutations can be huge to test thoroughly. In addition, the test suites used by SKI may be too small to explore an OS for bugs.

Razzer [39] combines static analysis with fuzzing for data race detection. In particular, Razzer first runs a points-to analysis across the whole kernel code base to identity potentially alias instruction pairs, *i.e.*, memory accesses that may point to the same memory location. After that, per each alias pair identified, Razzer tries to generate syscalls that reach the racy instructions at runtime. It does so with fuzzy syscall generation [32, 42], and sequential syscall traces are generated first. Once the alias relation is confirmed in the sequential execution, the trace is then paralleled into multi-threaded traces for actual data race detection.

Razzer presents an elegant pipeline for data race fuzzing, but it can be further improved: 1) running points-to analysis [71] on kernel file systems produces millions of may-alias pairs, which is almost impossible to enumerate one by one; 2) even for one alias pair, how to generate syscalls that may reach the racy instructions is less clear. KRACE aims to improve both aspects with the novel notion of alias coverage. Instead of pre-calculating the search space with points-to analysis, KRACE relies on coverage-guided fuzzing to expand the search in the concurrency dimension gradually. Analogically, this is similar to not enumerating every path in the control-flow graph but instead using an edge-coverage bitmap to capture the search progress. Doing so also eliminates the concern on how to generate syscalls that lead execution to specific locations.

2.3 Hunting Semantic and Logic Errors via Symbolic Execution

2.3.1 Symbolic execution in general

Symbolic execution is a popular program analysis technique introduced in the mid ’70s to test whether certain properties can be violated by a piece of software. Aspects of interest could be that no division by zero is ever performed, no NULL pointer is ever dereferenced, no backdoor exists that can bypass authentication, etc. While in general there is no automated way to decide some properties (*e.g.*, the target of an indirect jump), heuristics and approximate analyses can prove useful in practice in a variety of settings, including mission-critical and security applications.

In a concrete execution, a program is run on a specific input and a single control flow path is explored. Hence, in most cases concrete executions can only under-approximate the analysis of the property of interest. In contrast, symbolic execution can simultaneously explore multiple paths that a program could take under different inputs. This paves the road to sound analyses that can yield strong guarantees on the checked property. The key idea is to allow a program to take on symbolic—rather than concrete—-input values. Execution is performed by a symbolic executor, which maintains for each explored control flow path:

- a first-order Boolean formula that describes the conditions satisfied by the branches taken along that path, and
- a symbolic memory store that maps variables to symbolic expressions or values.

Branch execution updates the formula, while assignments update the symbolic store. A model checker, typically based on a satisfiability modulo theories (SMT) solver [72], is eventually used to verify whether there are any violations of the property along each explored path and if the path itself is realizable, *i.e.*, if its formula can be satisfied by some assignment of concrete values to the program’s symbolic arguments

From a theoretical perspective, (exhaustive) symbolic execution provides a *sound* and *complete* methodology for any decidable analysis. Soundness prevents false negatives, *i.e.*, all possible unsafe inputs are guaranteed to be found, while completeness prevents false positives, *i.e.*, input values deemed unsafe are actually unsafe. However, in practice, we often face challenges in achieving both soundness and completeness. Two important challenges are:

- *State space explosion*: how does symbolic execution deal with path explosion? Language constructs such as loops might exponentially increase the number of execution states. It is thus unlikely that a symbolic execution engine can exhaustively explore all the possible states within a reasonable amount of time.
- *Symbolic memory*: how does the symbolic engine handle pointers, arrays, or other complex objects? Code manipulating pointers and data structures may give rise not only to symbolic stored data, but also to addresses being described by symbolic expressions.

We will illustrate how these challenges can be systematically addressed in this thesis.

2.3.2 Symbolic execution tailored for OS kernels

With the recent advances in SMT solvers [72] such as Z3 [73] and CVC4 [74], symbolic execution has proven to be an effective technique in finding bugs in complex software applications [47, 48, 49, 75, 76]. Recent research has further made trade-offs between scalability and path coverage of symbolic execution. A few symbolic execution techniques are now able to analyze even OS kernels such as S2E [77] and FuzzBALL [78, 79]. In particular, S2E employs selective symbolic execution and relaxed execution consistency models to significantly improve the performance. A number of tools (e.g., SymDrive [29], Stack Spraying [80], and CAB-Fuzz [81]) built on top of S2E have been designed to analyze kernel code for various purposes.

Improvements by Deadline. DEADLINE also leverages the power of SMT solvers for *double-fetch bug* detection and uses a similar way to collect constraints and assign SR to variables as traditional symbolic executors. However, DEADLINE can be differentiated from them in two ways:

Path exploration strategy : DEADLINE performs path exploration offline and symbolically executes only within a particular path instead of exploring paths online by forking states whenever a conditional branch is encountered. This is because, unlike traditional symbolic executors whose primary goal is path discovery, DEADLINE is not bounded to execute the instructions that are irrelevant to the cause of a *double-fetch bug*, and DEADLINE takes full advantage of that by first filtering out these irrelevant instructions and then constructing paths that must go through at least two fetches and only checks along these paths.

Memory model : DEADLINE extends the memory model used in traditional symbolic executors in two aspects. 1) An epoch number is added to memory reads when they cross the kernel-user boundary to denote that different userspace fetches from the same address can be different, which is effectively the root cause of a *double-fetch bug*. 2) Instead of assuming a pointer can point to anywhere in the memory (*i.e.*, a flat linear array of bytes), DEADLINE keeps a mapping of pointers to memory objects and uses this to filter out *multi-reads* that are in fact unrelated fetches.

2.4 Other Approaches on Race Condition Detection

2.4.1 Static lockset analysis

Many works aim to find concurrency bugs with static analysis [62, 63, 64, 65, 66]. Most of these approaches rely on static lockset analysis and, hence, suffer from the high false-positive rate caused by missing the happens-before relation in the execution as well as the inherent limitations of the points-to analysis. For instance, RacerX [63] suffers from 50% false positives on the Linux kernel.

However, a major challenge in applying these works to data race detection in OS kernels is their lack of statefulness, *i.e.*, although extremely effective in finding bugs within one syscall execution, they miss bugs that occur because of the interaction between multiple syscalls, which happen to be the majority of cases in kernel operations.

2.4.2 Dynamic lockset and happens-before analysis

Happens-before + lockset tracking. Most of the initial works [82] found race conditions by relying on the *happens-before* analysis [83]. However, one of the prime issues with this approach is that it leads to false negatives. To improve the detection accuracy, Eraser [84] proposed the *lockset analysis*, in which users annotate the common lock/unlock methods and find atomicity violations. Later, several works [85, 86] proposed optimizations to either mitigate the overhead or minimize false positives. To further improve the effectiveness of dynamic data race detection, several works [87, 88] combined the idea of happens-before relation with lockset analysis.

Limited applicability in OS kernels Unfortunately, most of these works target userspace programs using simple synchronization primitives (*e.g.*, those provided by `pthread` or Java runtime), which only represent a small subset of synchronization mechanisms available in the Linux kernel. KRACE follows the same trend in combining happens-before and lockset analysis, but unlike prior works, KRACE provides a comprehensive framework that includes not only simple locking methods, such as pessimistic locks (*e.g.*, mutex, readers-writer lock, spinlock, etc.), but also optimistic locking protocols, such as sequence locks, and other forms of synchronization mechanisms that imply more than just mutual exclusion, *e.g.*, RCU [89] and other publisher-subscriber models.

Timing-based detection. Both lockset and happens-before analysis require code annotations and suffer from incompleteness, *i.e.*, a missing lock model leads to false positives. Several works overcome this issue with timing-based detection, *i.e.*, a thread is delayed for a certain duration at some memory accesses while the system observes whether there are conflicting accesses to the same memory during the delay [60, 61, 21]. Moreover, most of these works resort to sampling [88, 90, 91, 21, 61], as an optimization over completeness, to further minimize the runtime overhead caused by tracking memory accesses or code paths.

However, complete timing-based detection relies on precise control of thread execution speed and results in an enormous search space (both in where to delay and how long to delay), which again is not scalable in the kernel scope. As a result, in terms of race detection, KRACE resorts to a trial-and-error approach and fixes false positives introduced by ad-hoc mechanisms along with the development. Fortunately, due to the high coding standard and strict code review practice, ad-hoc synchronization is not common in kernel file systems.

CHAPTER 3

KRACE: GENERIC DATA RACE FUZZING FOR KERNEL FILE SYSTEMS

3.1 A Coverage Metric for Concurrent Programs

In this section, we show why branch coverage, the golden metric for fuzzing, might be insufficient to represent the exploration in the concurrency dimension, while at the same time, why alias coverage, our new proposal, fits this purpose.

3.1.1 Branch coverage for the sequential dimension

Branch coverage originates from the program control-flow graph (CFG), which is inherently a sequential view of program execution. As shown in Figure 3.1, in CFGs, execution flows through basic blocks and user-controllable inputs, *e.g.*, `size` in `SYS_truncate`, determine the set of edges that join the basic blocks. For a branch coverage-guided fuzzer: given an input (*e.g.*, a list of syscalls), it tracks the set of edges that are hit at runtime and leverages this feedback to decide whether this input is "useful" and should be kept for more mutations. Essentially, the more edges covered during execution, the more "useful" the input is. Intuitively, the fuzzer expects to probe more branches by further mutating the seed, and not surprisingly, once the branch coverage growth stalls, the fuzzer will shift its focus to other seeds for a more economical use of computing resources.

In the case shown in Figure 3.1, exhausting all branches sequentially will only yield the status of `B==1`, `B==2`, and `C==0`. After that, these execution paths (represented by the seeds covering them) will be de-prioritized and considered non-interesting by the fuzzer. However, this is not the end of the story. To trigger the data race when `B==C==2`, the execution of four critical instructions (`i1-i4`) has to be interleaved in a special way, as shown in Figure 3.2. Unfortunately, all six interleavings yield the same branch coverage, and the fuzzer is likely

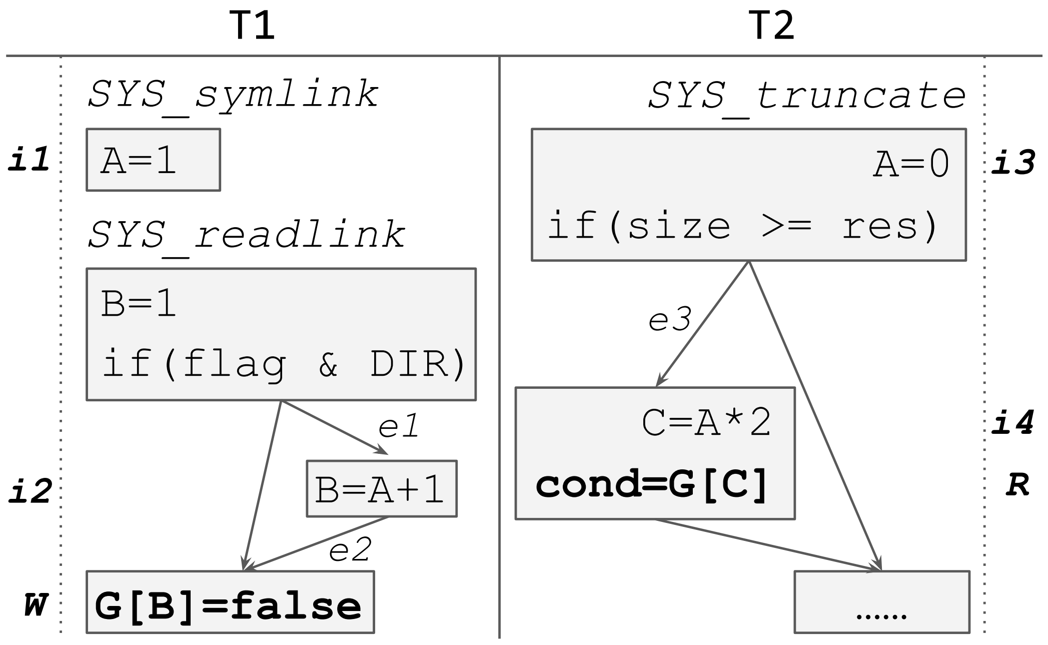


Figure 3.1: A data race found by KRACE when `symlink`, `readlink`, and `truncate` on the same `inode` run in parallel (simplified for illustration). The race is on the indexed accesses to a global array `G` and occurs only when `B==C`. `A` is lock-protected. This is one example showing branch coverage is not sufficient in approximating execution states of highly concurrent programs. It is not difficult to cover all branches in this case with existing fuzzers, but to trigger the data race, merely covering branches `e1-e3` is not enough. The thread interleavings between four instructions `i1-i4` are equally important. The valid interleavings that may trigger the data race are shown in Figure 3.2.

to give up the seed upon hitting a few of them.

Further note that this is an extremely simplified example that involves only six possible interleavings among two threads. In actual executions, the concurrency dimension can be huge, as the instructions executed by each thread are usually in the thousands or even millions, while there will be tens of threads running at the same time. As a result, when fuzzing highly concurrent programs, we need to pay attention to not only code paths explored, but also meaningful thread interleavings explored that yield to the same branch coverage. In other words, if the fuzzer believes that there could be unexplored thread interleavings in a seed, the seed should not be de-prioritized.

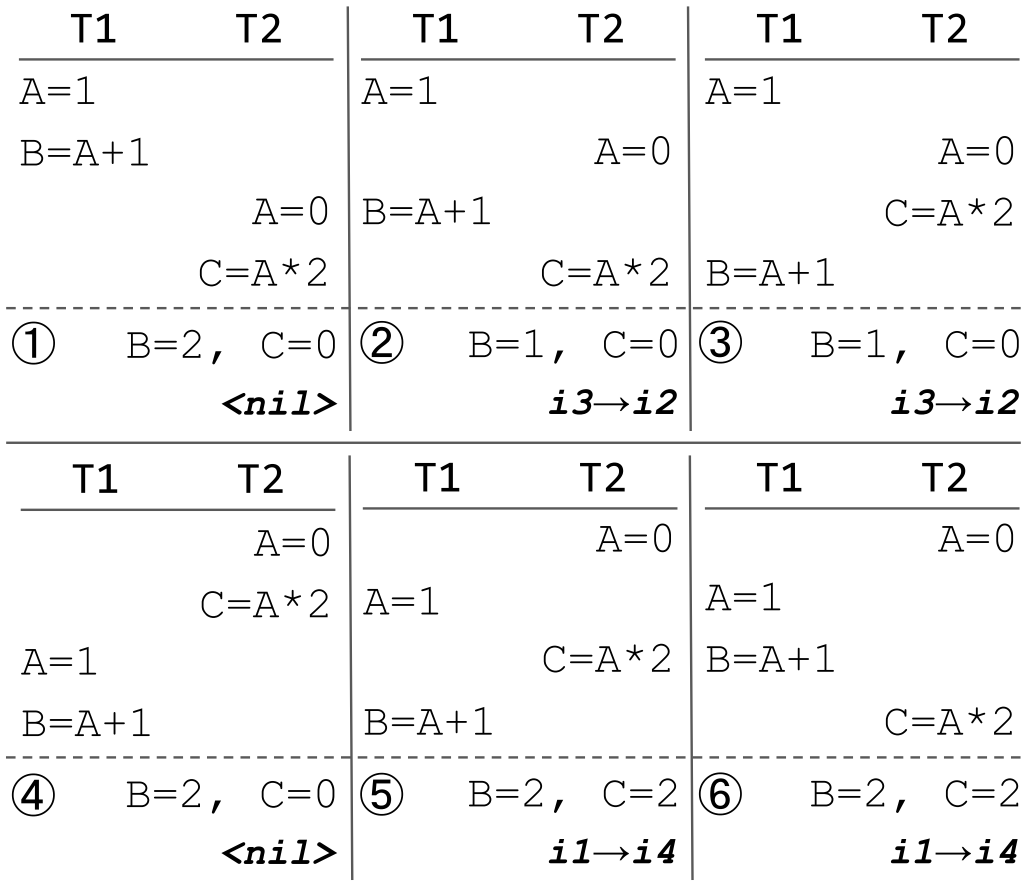


Figure 3.2: Possible thread interleavings among the four instructions shown in Figure 3.1. Out of the 6 interleavings, only 3 interleavings (①/④, ②/③, ⑤/⑥) are effective depending on `A`'s value when `B` and `C` read it. Each effective interleaving results in different alias coverage. Only ⑤/⑥ may trigger the data race.

3.1.2 Alias coverage for the concurrency dimension

Intuition. At first thought, recording the exploration of thread interleavings can be futile. A realistic kernel file system at its peak time may use over 60 internal threads, where each thread may execute over 100,000 instructions. The total possible number of thread interleavings is 60^{100000}, an enormous search space that no bitmap can ever approximate.

However, it is worth noting that not all interleaved executions are useful. In fact, only interleavings of memory-accessing instructions to the same memory address matters. As shown in Figure 3.1, interleaving instructions apart from `i1-i4` has no effect on the final results of `B`, `C`, as well as the manifestation of the data race. This is true in the actual code, where hundreds and thousands of instructions sit between `i1`, `i3` and `i2`, `i4`.

In other words, based on the crucial observation that data races, and even in the broader term, concurrency bugs, typically involve unexpected interactions among a few instructions executed by a small number of threads [92, 93, 18], if KRACE is able to track how many interactions among these few memory-accessing instructions have been explored, it is sufficient to represent thread interleaving coverage and to find data races. This is precisely what gets tracked by alias coverage.

A formal definition. First, suppose all memory-accessing instructions in a program are uniquely labeled: `i1`, `i2`, ..., `iN`. At runtime, each memory address `M` keeps track of its last *define operation*, *i.e.*, the last instruction that writes to it as well as the context (thread) that issues the write, represented by `A ← <ix, tx>`. Now, in the case in which a new access to `M` is observed, carried by instruction `iy` from context `ty`: if `iy` is a write instruction, update `A ← <iy, ty>` to reflect the fact that `A` is redefined. Otherwise:

- if `tx == ty`, *i.e.*, same context memory access, do nothing,
- or else, record *directed* pair `ix→iy` in the alias coverage.

Figure 3.2 is a working example of this alias coverage tracking rule. In cases ① and ④, there is no inter-context define-then-use of memory address `A`, and hence, the alias coverage map is empty. On the other hand, in cases ② and ③, the calculation of `B` in `T1` relies on `A` defined in `T2`, hence the pair `i3→i2`. The same rule applies to cases ⑤ and ⑥.

Feedback mechanism. Essentially, alias coverage provides a signal to the fuzzer on whether it should expect more useful thread interleavings out of the current test case, *i.e.*, a multi-threaded syscall sequence. If the alias coverage keeps growing, the fuzzer should come up with more delay schedules to inject at the memory-accessing instructions (detailed in subsection 3.2.2) in the hope of probing unseen interleavings. Otherwise, if the coverage growth stalls, it is a sign that the concurrency dimension of the current test case is toward saturation, and the most economical choice is to switch to other seeds for further exploration.

Coverage sensitivity fine-tuning. Finding one-suits-all coverage criteria has been a never-ending quest in software engineering [94]. Even the branch coverage has several variations, such as N-gram branch coverage, context-sensitive coverage [52], etc., which are well-documented and compared in a recent survey [95]. However, despite the fact that branch coverage is always subsumed by program whole-path coverage, branch coverage is still preferred over path coverage, as the latter is overly sensitive to input changes and thus requires a much larger bitmap to hold and compare. On the other hand, branch coverage strikes a balance among effectiveness, execution speed, and bitmap accounting overhead.

Similarly, alias coverage strives to find such a balance point in the concurrency dimension. In our experiments with kernel file system fuzzing, KRACE observed 63,590 unique pairs of alias instructions (directed access). Based on the data, for an empirical estimation, a bitmap of size 128KB should be sufficient to avoid heavy collisions, which is close to AFL's branch coverage bitmap size (64KB). In addition, if more sensitivity is needed for alias coverage, KRACE can be easily adopted from 1st-order alias pair (alias coverage) to 2nd-order alias pair, Nth-order alias pair, and up-to total interleaving coverage. This is not planned in KRACE but could be a promising extension for future exploration.

3.2 Input Generation for Concurrency Fuzzing

In this section, we present how to synthesize and merge multi-threaded syscall sequences for file system fuzzing, as well as how to exploit a hidden input domain—thread delay schedule—to accelerate thread interleaving probing.

3.2.1 Multi-threaded syscall sequences

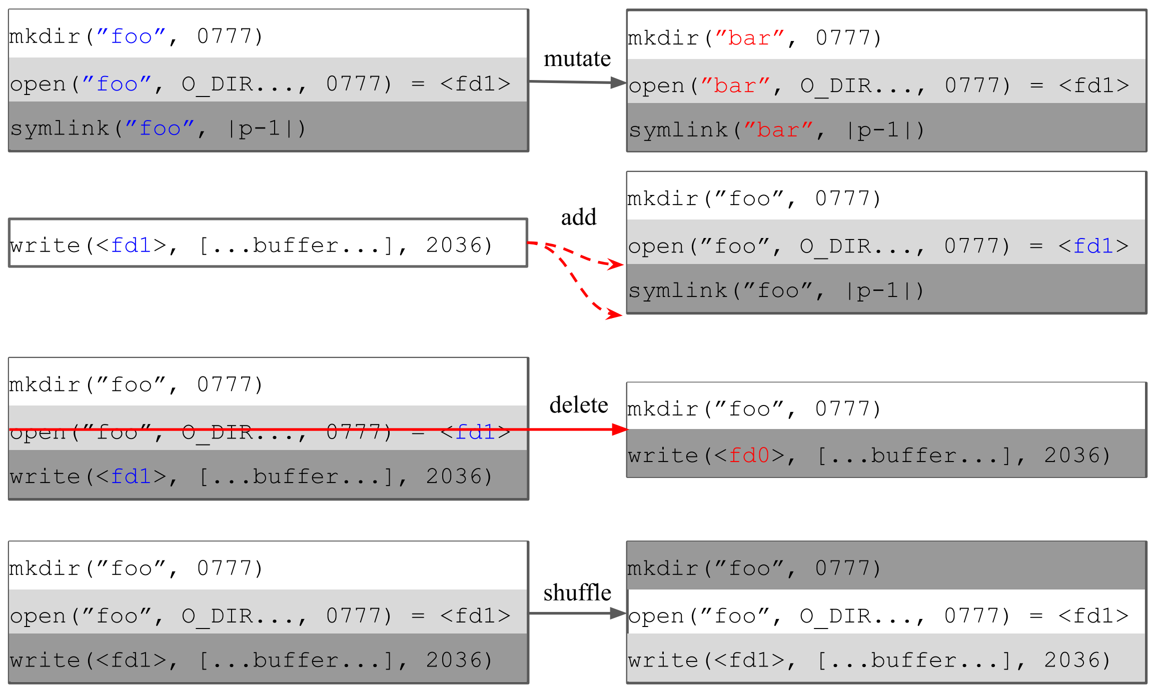
Specification-based synthesization. The goal of syscall generation and mutation is to generate diverse and complex file operations that are otherwise difficult for human developers to contemplate. Given that syscalls are highly structured data, it is almost fruitless to mutate their arguments blindly. As a result, we use a specification to guide the generation and mutation of syscall arguments. A feature worth highlighting in KRACE's specification is the encoding of inter-dependencies among syscalls, especially path components and file descriptors (fd), which are most relevant to file system fuzzing. To illustrate, as shown in Figure 3.4, the `open` syscall in seed 1 reuses the same path component in the `mkdir` syscall, while the `write` syscall in seed 2 relies on the return value of `creat`.

Seed format. The seed input for KRACE is a multi-threaded syscall sequence. Internally, it is represented by a single list of syscalls (*a.k.a*, the main list) and a configurable number of sub-lists (3 in KRACE) in which each sub-list contains a disjoint sequence of syscalls in the main list. Each sub-list represents what will be executed by each thread at runtime. To illustrate, as shown in Figure 3.4, seed 1 has three threads, where each thread will be executing `mkdir-close`, `mknod-open-close`, and `dup2-symlink`, respectively, marked in different greyscale.

Evolution strategies. KRACE uses four strategies to evolve a seed for both branch and alias coverage, as shown in Figure 3.3.

- **Mutation**: a randomly picked argument in one syscall will be modified according to specification. If a path component is mutated, it is cascaded to all its dependencies.
- **Addition**: a new syscall can be added to any part of the trace in any thread, but must be after its origins.
- **Deletion**: a random syscall is kicked out of the main list and the sub-list. In case a file descriptor is deleted, its dependencies are forced to re-select another valid file.
- **Shuffling**: syscalls in the main list are redistributed to sub-lists, but their orders in the main list are preserved.

Merging multi-threaded seeds. The power of fuzzing lies not only in evolving a single seed but also in joining two seeds to produce more interesting test cases. To enable seed merging in KRACE, a naive solution might be simply to concatenate two traces. However, this is not the most economical use of seeds, as it forgoes the opportunities to find new coverage by further interleaving these high-quality executions.



```
mkdir("foo", 0777)
open("foo", O_DIR..., 0777) = <fd1>
symlink("foo", |p-1|)

mkdir("bar", 0777)
open("bar", O_DIR..., 0777) = <fd1>
symlink("bar", |p-1|)

write(<fd1>, [...buffer...], 2036)

mkdir("foo", 0777)
open("foo", O_DIR..., 0777) = <fd1>
symlink("foo", |p-1|)

mkdir("foo", 0777)
~~open("foo", O_DIR..., 0777) = <fd1>~~
write(<fd1>, [...buffer...], 2036)

mkdir("foo", 0777)
write(<fd0>, [...buffer...], 2036)

mkdir("foo", 0777)
open("foo", O_DIR..., 0777) = <fd1>
write(<fd1>, [...buffer...], 2036)

mkdir("foo", 0777)
open("foo", O_DIR..., 0777) = <fd1>
write(<fd1>, [...buffer...], 2036)
```

Figure 3.3: Illustration of four basic syscall sequence evolution strategies supported in KRACE: mutation, addition, deletion, and shuffling. For KRACE, each seed contains multi-threaded syscall sequences and each thread trace is highlighted in different shades of greyscale.

KRACE adopts a more advanced merging scheme: upon merging, the main lists of the two seeds are interweavingly joined, *i.e.*, the relative orders of syscalls are still preserved in the resulting main list as well as in the sub-lists. As a result, the syscall inter-dependencies are preserved too. As shown in Figure 3.4, all the dependencies on path and fds are properly preserved after merging (highlighted in corresponding colors).

Primitive collection. Successful syscalls are valuable assets out of the file system fuzzing practice, not only because they lead to significantly broader coverage than failed syscalls, but also because they can be difficult, and sometimes even fortunate, to generate due to the dependencies among them. This is true especially for long traces of closely related syscalls. As a result, upon discovering a new seed, KRACE first prunes it and retains only successful syscalls and further splits these syscalls into non-disjoint primitives where each primitive is self-contained, *i.e.*, for any syscall, all its path and fd dependencies (also syscalls) are captured in the same primitive.

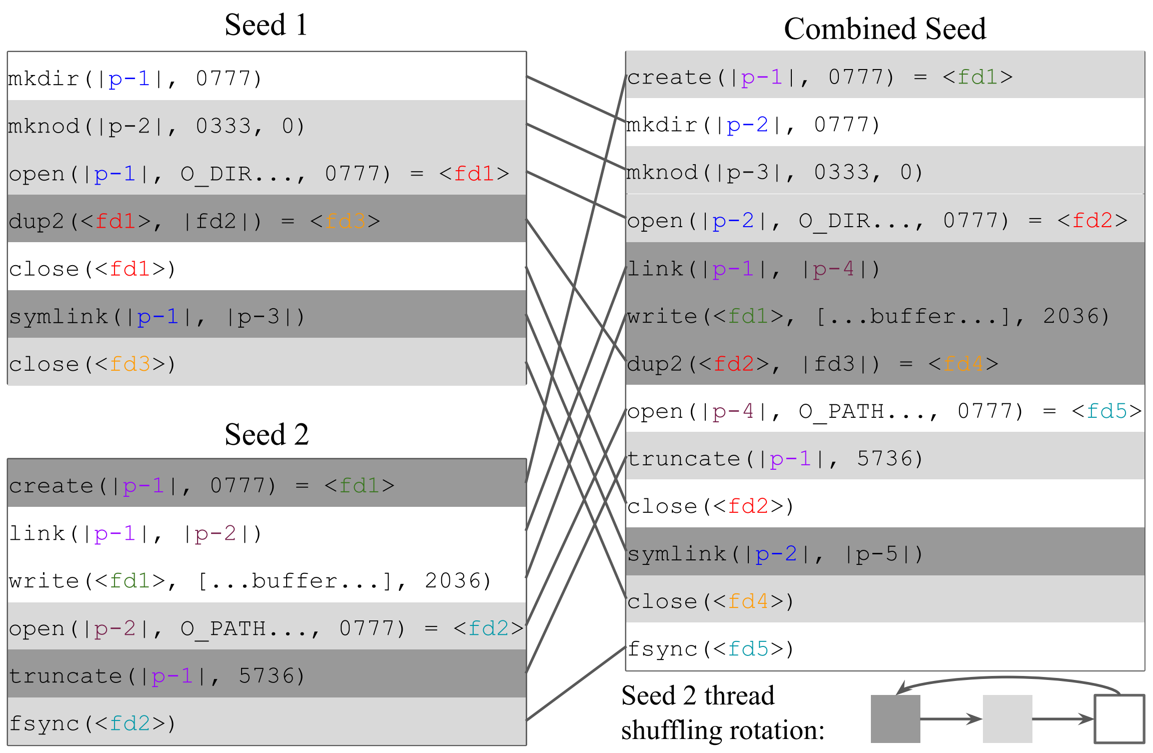

Seed 1
mkdir(|p-1|, 0777)
mknod(|p-2|, 0333, 0)
open(|p-1|, O_DIR..., 0777) = <fd1>
dup2(<fd1>, |fd2|) = <fd3>
close(<fd1>)
symlink(|p-1|, |p-3|)
close(<fd3>)
Seed 2
create(|p-1|, 0777) = <fd1>
link(|p-1|, |p-2|)
write(<fd1>, [...buffer...], 2036)
open(|p-2|, O_PATH..., 0777) = <fd2>
truncate(|p-1|, 5736)
fsync(<fd2>)
Combined Seed
create(|p-1|, 0777) = <fd1>
mkdir(|p-2|, 0777)
mknod(|p-3|, 0333, 0)
open(|p-2|, O_DIR..., 0777) = <fd2>
link(|p-1|, |p-4|)
write(<fd1>, [...buffer...], 2036)
dup2(<fd2>, |fd3|) = <fd4>
open(|p-4|, O_PATH..., 0777) = <fd5>
truncate(|p-1|, 5736)
close(<fd2>)
symlink(|p-2|, |p-5|)
close(<fd4>)
fsync(<fd5>)
Seed 2 thread shuffling rotation:

Figure 3.4: Semantic-preserving combination of two seeds. For KRACE, each seed contains multi-threaded syscall sequences and each thread trace is highlighted in different shades of greyscale.

Over the course of fuzzing, KRACE has accumulated a pool of around 10,000 primitives covering 68 file system related syscalls for which KRACE has a specification. In each primitive, file operations span across 3 threads, with each thread containing 1-10 syscalls, and most importantly, all syscalls succeed. We will open-source this collection in the hope that these primitives may serve as quality seeds for future concurrent file system fuzzing.

3.2.2 Thread scheduling control (weak form)

Thread scheduling is a hidden input domain for concurrency programs. Unfortunately, there is no way to control kernel scheduling by merely mutating syscall traces. Hooking the scheduling implementation (or using a hypervisor) and systematically permuting the schedules might be possible for small-scale programs [93] or for a few user threads in the kernel [18, 39]. But these algorithms are far from being scalable enough to cover all kernel threads. For a taste of the scalability requirement, Figure 3.5 shows the level of concurrency

```
struct btrfs_fs_info {
    /* work queues */
    struct btrfs_workqueue *workers;
    struct btrfs_workqueue *delalloc_workers;
    struct btrfs_workqueue *flush_workers;
    struct btrfs_workqueue *endio_workers;
    struct btrfs_workqueue *endio_meta_workers;
    struct btrfs_workqueue *endio_raid56_workers;
    struct btrfs_workqueue *endio_repair_workers;
    struct btrfs_workqueue *rmw_workers;
    struct btrfs_workqueue *endio_meta_write_workers;
    struct btrfs_workqueue *endio_write_workers;
    struct btrfs_workqueue *endio_freespace_worker;
    struct btrfs_workqueue *submit_workers;
    struct btrfs_workqueue *caching_workers;
    struct btrfs_workqueue *readahead_workers;
    struct btrfs_workqueue *fixup_workers;
    struct btrfs_workqueue *delayed_workers;
    struct btrfs_workqueue *scrub_workers;
    struct btrfs_workqueue *scrub_wr_completion_workers;
    struct btrfs_workqueue *scrub_parity_workers;
    struct btrfs_workqueue *qgroup_rescan_workers;
    /* background threads */
    struct task_struct *transaction_kthread;
    struct task_struct *cleaner_kthread;
};
```

Figure 3.5: 20 work queues and 2 background threads used by `btrfs`. This does not cover all asynchronous activities observable at runtime.

introduced by the `btrfs` module alone, not to mention other background threads forked by the block layer, loop device, timers, and RCU.

Runtime delay injection. KRACE resorts to delay injection to achieve a weak (and indirect) control of kernel scheduling, based on the observation that only shared memory accesses matter in thread interleavings. KRACE's delay injection scheme is extremely simple, as shown in Figure 3.6. Before launching the kernel, KRACE generates a ring buffer of random numbers and maps it to the kernel address space. At every memory access point, the instrumented code fetches a random number from the ring buffer, say `T`, and delays for `T` memory accesses observed by KRACE system-wise (*i.e.*, in other threads).

A ring buffer is used to hold the random numbers, as KRACE cannot pre-determine how many injection points are needed for each execution, not to mention that such a number may be extremely large. Injecting delays at memory access points is at the finest granularity for delay injection. Although this works well in file system fuzzing, it might nevertheless be too fine-grained and introduces too much overhead. The injection points can be at the granularity

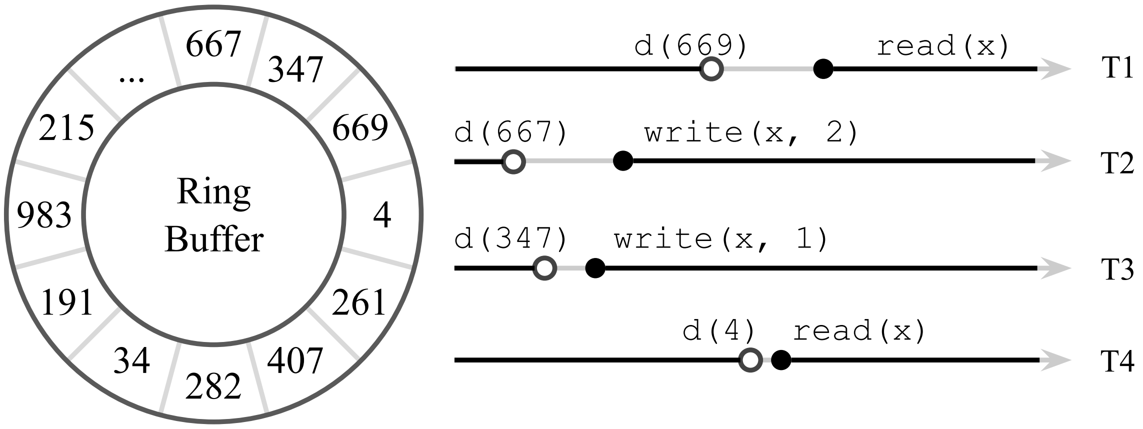


Figure 3.6: The delay injection scheme in KRACE. In this example, white and black circles represent the memory access points before and after delay injection. Injecting delays uncovers new interleavings in this case, as the read and write order to the memory address `x` is reversed.

of basic blocks or functions or even customized locations such as locking operations, etc.

3.3 A Data Race Checker for Kernel Complexity

Although the definition of data races is simple, finding them in a kernel execution trace can be difficult, primarily because of the variety of synchronization primitives available in the kernel code base as well as the ad-hoc mechanisms implemented by each individual file system. In this section, we enumerate the major categories of kernel synchronization primitives and describe how they can be modeled in KRACE.

3.3.1 Data race detection procedure

Overview. We say a pair of memory operations, `<ix, iy>`, is a data race candidate if, at runtime, we observed that

- they access the same memory location,
- they are issued from different contexts `tx` and `ty`,
- at least one of them is a write operation.

Such information is trivial to obtain dynamically by simply hooking every memory access. The difficulty lies in confirming whether a data race candidate is a true race. For this, we need two more analysis steps to check that:

- no locks are commonly held by both contexts, `tx` and `ty`, at the time when memory operations `ix` and `iy` are issued from them, respectively. [lockset (subsection 3.3.2)]
- no ordering between `ix` and `iy` can be inferred based on the execution: *i.e.*, there is no reason `ix` *must happen-before* `iy` or the other way around, regardless of how `tx` and `ty` are scheduled. [happens-before (subsection 3.3.3)]

Conceptually, lockset analysis produces no false negatives, *i.e.*, if there is a data race in the execution trace, it is guaranteed to be flagged by the lockset analysis. But lockset analysis is prone to false positives, as it ignores the ordering information. Happens-before analysis helps in filtering these false positives.

Kernel complexity. Although conceptually simple, lockset analysis requires a complete model of all locking mechanisms available in the kernel, and similarly, happens-before analysis requires all thread ordering primitives to be annotated. Otherwise, false positives will arise. However, after nearly 30 years of development, the Linux kernel has accumulated a rich set of synchronization mechanisms. KRACE takes a best-effort approach in modeling all major synchronization primitives as well as ad-hoc ones if we encounter them in our experiment. Some representative ad-hoc schemes modeled by KRACE are presented in subsection 3.3.4.

3.3.2 Lockset analysis

Most kernel locking primitives differentiate between reader and writer roles. The major difference is that a reader-lock can be acquired by multiple threads at the same time, as long as its corresponding writer-lock is not held; while a writer-lock can only be held by at most one thread. KRACE follows this distinction and tracks the acquisitions and releases of both reader- and writer-locks for each thread at runtime. Formally, such information is stored in the form of a lockset: denoted by $LS^{R}_{<t,i>}$ for the reader-side lockset for thread `t` at instruction `i` as well as $LS^{W}_{<t,i>}$ for the writer-side lockset. Both locksets are cached and attached to a memory cell whenever a memory access on that thread is observed, as shown in Figure 3.7.

The lockset analysis is simple as the following: for each data race candidate `<tx, ix>` and `<ty, iy>`, if any of the following conditions holds, this candidate is not a true race.

$$LS^{R}_{<tx,ix>} \cap LS^{W}_{<ty,iy>} \neq \varnothing \tag{3.1}$$

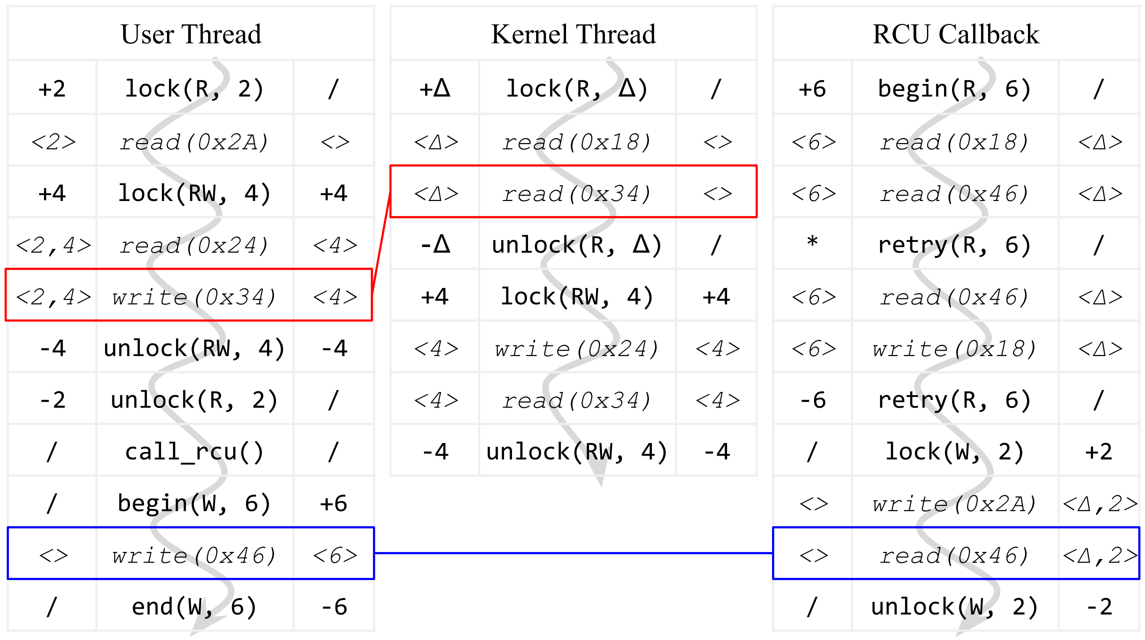
$$LS^{W}_{<tx,ix>} \cap LS^{R}_{<ty,iy>} \neq \varnothing \tag{3.2}$$

$$LS^{W}_{<tx,ix>} \cap LS^{W}_{<ty,iy>} \neq \varnothing \tag{3.3}$$

On the other hand, if none of the conditions hold for a data race candidate, then the execution of `tx` and `ty` can be interleaved without restrictions around those memory accesses, as shown in the reading and writing of addresses `0x34` and `0x46` in Figure 3.7, hence, leading to data races.

Pessimistic locking. Most of the kernel locking primitives are pessimistic locking, *i.e.*, whoever tries to acquire the lock will be blocked from further execution until the lock holder releases it. As a result, their APIs are always in pairs of `lock` and `unlock` to mark the start and end of a critical section. Examples of such locks include spin lock, reader/writer spin lock, mutex, reader/writer semaphore, and bit locks.

A slightly trickier primitive is the RCU lock, in which only reader-side critical sections are marked with `rcu_read_[un]lock` and the writer-side critical section is not marked



User Thread		
+2	lock(R, 2)	/
<2>	*read(0x2A)*	<>
+4	lock(RW, 4)	+4
<2,4>	*read(0x24)*	<4>
<2,4>	*write(0x34)*	<4>
-4	unlock(RW, 4)	-4
-2	unlock(R, 2)	/
/	call_rcu()	/
/	begin(W, 6)	+6
<>	*write(0x46)*	<6>
/	end(W, 6)	-6

Kernel Thread		
+Δ	lock(R, Δ)	/
<Δ>	*read(0x18)*	<>
<Δ>	*read(0x34)*	<>
-Δ	unlock(R, Δ)	/
+4	lock(RW, 4)	+4
<4>	*write(0x24)*	<4>
<4>	*read(0x34)*	<4>
-4	unlock(RW, 4)	-4

RCU Callback		
+6	begin(R, 6)	/
<6>	*read(0x18)*	<Δ>
<6>	*read(0x46)*	<Δ>
*	retry(R, 6)	/
<6>	*read(0x46)*	<Δ>
<6>	*write(0x18)*	<Δ>
-6	retry(R, 6)	/
/	lock(W, 2)	+2
<>	*write(0x2A)*	<Δ,2>
<>	*read(0x46)*	<Δ,2>
/	unlock(W, 2)	-2

Figure 3.7: Illustration of lockset analysis in KRACE. This example shows almost all locking mechanisms commonly used in the kernel, including
1) spin lock and mutexes—`[un]lock(RW, -)`,
2) reader/writer lock—`[un]lock(R/W, *)`,
3) RCU lock—specially denoted with symbol Δ, and
4) sequence lock—`begin/end/retry(R/W, *)`.
The left column shows the content in the reader lockset at the time of memory operation or changes to the lockset caused by other operations (`/` denotes no change). The right column shows the writer counterpart. The two data races are highlighted in red and blue squares.

by any lock/unlock APIs, instead, it is guaranteed by the RCU grace period waiting. More specifically, when `__rcu_reclaim` schedules an RCU callback into execution, it is guaranteed that there is no RCU reader-side critical section running. Hence, in KRACE, we hook the RCU callback dispatcher and mark RCU writer lock and unlock before and after the callback execution.

Optimistic locking. The Linux kernel is gradually shifting toward lock-free design and the most prominent evidence in recent years is the wide adoption of sequence locks [96]. A sequence lock is, in fact, more similar to a transaction than to a conventional lock. The reader is allowed to run optimistically into the critical section, hoping that the data it reads will not be modified during the transaction (hence the optimism), and aborts and retries if the data does get modified.

While boosting performance, a challenge brought by the sequence lock is that there is no clear end of the reader-side critical section. As shown in Figure 3.7, after a transaction `begins`, the `retry` can be called multiple times, perhaps one for mid-of-progress checking and the other one for before-commit checking; in theory, each `retry` could be an `unlock`-equivalent that marks the end of the critical section. If the lockset analysis is performed online (*i.e.*, during execution), the lockset states should fork to capture that the `retry` may or may not be an `unlock`. For KRACE, since it uses offline lockset analysis, it may simply read the execution trace ahead to know whether there are more `retries` and behave correspondingly.

3.3.3 Happens-before analysis

Intuitively, happens-before analysis tries to find the causal relations between specific execution points in the threads. For example, a kernel thread only gets into running if another thread forks it; as a result, there is no way to schedule the spawned thread before the parent thread creates it. This implies that whatever happens before the thread creation points cannot be data racing against anything in the spawned thread. In the example shown in Figure 3.8, there is no way for `i2` to be racing against `i6`, as without queuing the work on the work queue (`c2`→`c8`), `i6` won't even be executed in the first place. Similarly, scheduling a thread that is waiting for a condition to be true will not make it run bypassing the barrier. Therefore, it is not possible for `i4` to race against `i8`, as only when the `wake_up` call is reached (`c12`→`c5`) can `i4` be executed.

This intuition shows how a happens-before relation can be formally checked: by hooking kernel synchronization APIs, *e.g.*, when a callback function is queued and when it is executed, we could find the synchronization points (nodes) between threads as well as the causality events (represented by edges), as shown in Figure 3.8. Since the nodes in one thread are already inherently connected according to program order, the whole execution becomes a directed acyclic graph. Consequently, determining whether two points, `<tx, ix>` and

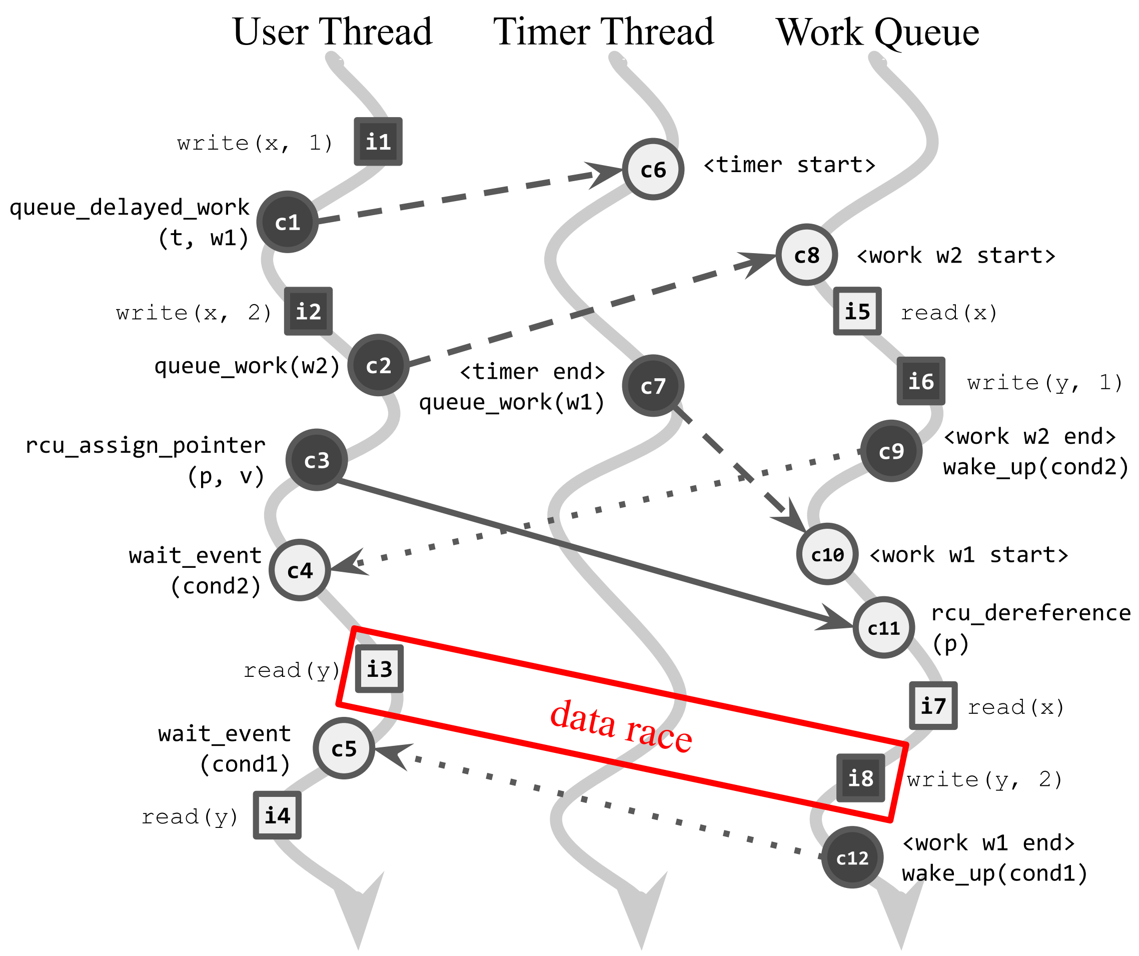


Figure 3.8: Illustration of happens-before reasoning in KRACE. This example shows a very typical execution pattern in kernel file systems where the user thread schedules two asynchronous works on the work queue and checks for their results later in the execution. In particular, one of the asynchronous works is a delayed work that also goes through the timer thread. Fork-style, join-style, and publisher-subscriber relations are represented by dashed, dotted, and solid arrows, respectively. The only data race is highlighted in the red square.

`<ty, iy>`, may race is translated into a graph reachability problem. If a path exists from `<tx, ix>` to `<ty, iy>`, it means that point `X` *happens-before* `Y` and thus cannot be racing. The same applies if we can establish `Y` *happens-before* `X`. On the other hand, if no such path can be found, a happens-before relation cannot be established and the pair should be flagged, as in the case of `i3` and `i8`. All other accesses are reachable in the graph, and hence, they cannot be racing even without lock protections.

The happens-before relation commonly found in kernel file systems can be broadly categorized into three types:

Fork-style relations include RCU callbacks registered with `call_rcu`, work queues and `kthread`-simulated work queues, direct `kthread` forking, timers, software interrupts (`softirq`), as well as inter-processor interrupts (`IPI`). Hooking their kernel APIs is as easy as finding corresponding functions that register the callback and dispatch the callback.

Join-style relations include the completion API and a wide variety of `wait_*` primitives such as `wait_event`, `wait_bit`, and `wait_page`. Hooking their kernel APIs requires locating their corresponding `wake_up` calls besides the `wait` calls.

Publisher-subscriber model mainly refers to the RCU pointer assignment and dereference procedure [89]. For example, if one user thread retrieves a file descriptor (`fd`) from the `fdtable` which is RCU-guarded, the new `fd` must have been published first, hence the causality ordering. The object allocate-and-use pattern also falls into this realm: the publisher thread allocates memory spaces for an object, initializes its fields, and inserts the pointer to a global or heap-based data structure (usually a list or hashtable), while the subscriber thread later dereferences the pointer and uses the object. As a result, KRACE also tracks the memory allocation APIs and monitors when the allocated pointer is first stored into a public memory slot and when it is used again to establish the ordering automatically.

3.3.4 Ad-hoc synchronization schemes in kernel file systems

Although ad-hoc synchronization schemes are considered harmful [97], they may still exist in kernel file systems for performance or functionality enhancements. Whenever we encounter an ad-hoc scheme (usually when analyzing false positives), we annotate it in the same way as major synchronization APIs so that subsequent runs will not report the false data races caused by it. In this section, we present two examples we encountered in `btrfs`.

Ad-hoc locking. An ad-hoc lock has two implications: 1) there will be data races in the lock implementation and these data races are all benign races; and 2) lock internals should be abstracted in a way that the lockset analysis can easily understand. A representative example is the `btrfs` tree lock, and the purpose of having the tree lock is to be convertible

```
1  /* acquire a spinning write lock, wait for both
2   * blocking readers or writers */
3 void btrfs_tree_lock(struct extent_buffer *eb)
4 {
5     u64 start_ns = 0;
6     if (trace_btrfs_tree_lock_enabled())
7         start_ns = ktime_get_ns();
8
9     WARN_ON(eb->lock_owner == current->pid);
10 again:
11     wait_event(eb->read_lock_wq,
12             atomic_read(&eb->blocking_readers) == 0);
13     wait_event(eb->write_lock_wq, eb->blocking_writers == 0);
14     write_lock(&eb->lock);
15     if (atomic_read(&eb->blocking_readers)
16             || eb->blocking_writers) {
17         write_unlock(&eb->lock);
18         goto again;
19     }
20     btrfs_assert_spinning_writers_get(eb);
21     btrfs_assert_tree_write_locks_get(eb);
22     eb->lock_owner = current->pid;
23 }
24 /* drop a spinning or a blocking write lock. */
25 void btrfs_tree_unlock(struct extent_buffer *eb)
26 {
27     int blockers = eb->blocking_writers;
28     BUG_ON(blockers > 1);
29
30     btrfs_assert_tree_locked(eb);
31     eb->lock_owner = 0;
32     btrfs_assert_tree_write_locks_put(eb);
33
34     if (blockers) {
35         btrfs_assert_no_spinning_writers(eb);
36         eb->blocking_writers--;
37         cond_wake_up(&eb->write_lock_wq);
38     } else {
39         btrfs_assert_spinning_writers_put(eb);
40         write_unlock(&eb->lock);
41     }
42 }
```

Figure 3.9: A snippet of the `btrfs` tree lock (writer side only).

between blocking and non-blocking mode, as shown in Figure 3.9.

In these functions, almost every memory access to the fields in the extent buffer, `eb`, could be racing against other accesses. *e.g.*, `eb->lock_owner` at line 12 against `eb->lock_owner = 0` at line 40. So the first annotation for KRACE is to assume all data races within these functions are safe and benign races.

To further encode the locking semantics for lockset analysis, we study the tree lock APIs and map their functionality into a simple reader-writer lock format as shown in Table 3.1. In other words, calling the, *e.g.*, `tree_lock` will be treated equally as calling the writer-

Table 3.1: Semantic mapping between the tree lock and conventional locks (in particular, the readers-writer lock).

Tree lock API	Lockset mapping
`tree_lock`	`writer-lock`
`tree_unlock`	`writer-unlock`
`tree_read_lock`	`reader-lock`
`tree_read_lock_atomic`	`reader-lock`
`tree_read_unlock`	`reader-unlock`
`tree_read_unlock_blocking`	`reader-unlock`
`tree_set_lock_blocking_read`	`no-op if read-locked`
`tree_set_lock_blocking_write`	`no-op if write-locked`
`try_tree_read_lock`	`reader-lock if succeed`
`try_tree_write_lock`	`writer-lock if succeed`

lock in the conventional locking mechanisms. Although `tree_lock` performs much more computation (*e.g.*, waiting for both blocking and non-blocking readers), from the lockset perspective, it is equivalent to a writer-lock.

Ad-hoc ordering. Ad-hoc ordering implies undocumented casual relations between thread executions and a good example is the customization of the conventional kernel work queue in `btrfs`, as shown in Figure 3.10.

In this example, the `set_bit` and `test_bit` (line 17 and 23), establish an additional causal relation beyond the normal `queue_work` semantic: the ordered function only gets into execution when the normal function finishes. Thus, although the observed happens-before relation is line 8 $\rightarrow$ line 24 and line 11 $\rightarrow$ line 15, the actual relation is line 8 $\rightarrow$ line 11 $\rightarrow$ line 15 $\rightarrow$ line 24.

3.4 Putting Everything Together

3.4.1 Architecture

Figure 3.11 shows the overall architecture of KRACE. The primary purpose of having the compile-time preparation is to embed a KRACE runtime into the kernel such that alias coverage (as well as branch coverage) can be collected dynamically. The runtime is also

```c
static inline void __btrfs_queue_work(struct __btrfs_workqueue *wq,
        struct btrfs_work *work)
{
    unsigned long flags;
    work->wq = wq;
    if (work->ordered_func) {
        spin_lock_irqsave(&wq->list_lock, flags);
        list_add_tail(&work->ordered_list, &wq->ordered_list);
        spin_unlock_irqrestore(&wq->list_lock, flags);
    }
    queue_work(wq->normal_wq, &work->normal_work);
}
static void normal_work_helper(struct btrfs_work *work) {
    /* ... */
    work->func(work);
    if (need_order)
        set_bit(WORK_DONE_BIT, &work->flags);
    /* ... */
}
static void run_ordered_work(struct __btrfs_workqueue *wq) {
    /* ... */
    work = list_entry(list->next, struct btrfs_work, ordered_list);
    if (test_bit(WORK_DONE_BIT, &work->flags))
        work->ordered_func(work);
    /* ... */
}
```

Figure 3.10: A snippet of the `btrfs` work queue implementation.

responsible for collecting information for data race checking, leveraging the kernel API hooking. On the other hand, the fuzzing loop is still conventional, covering seed selection, mutation, and execution, with the exception that in KRACE, a test case is considered "interesting" as long as new progress is found in either of the coverage bitmaps. In addition, all components are updated to handle the new seed format for concurrency fuzzing: multi-threaded syscall sequences.

Code instrumentation. Since the focus of KRACE is file systems, we only instrument memory access instructions in the target file system module and its related components such as the virtual file system layer (`VFS`) or the journaling module, *e.g.*, `jbd2` for `ext4`. On the other hand, API annotations are performed on the main kernel code base and have an effect even when the execution goes out of the functions in our target file system: the locks acquired and released, as well as the ordering primitives (*e.g.*, queuing a timer), will be faithfully recorded. In this way, KRACE does not suffer from false positives in cases like block layer calls into a callback in the file system layer but we do not know the prior locking contexts.

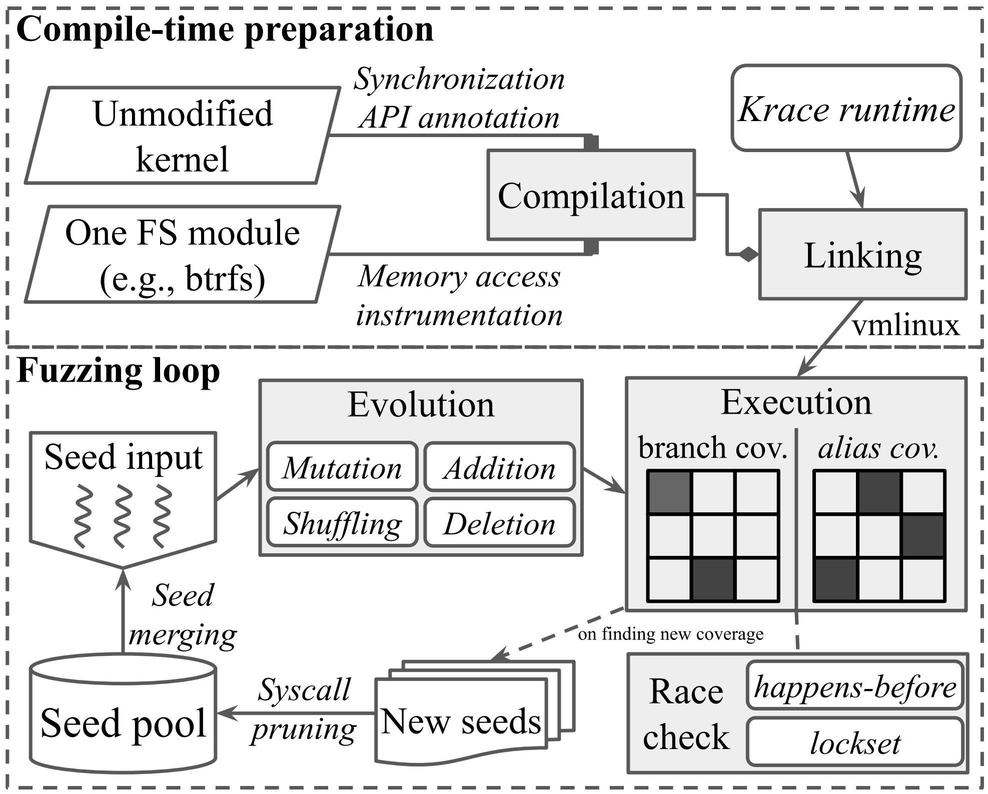


Figure 3.11: An overview of KRACE's architecture and major components. Components in *italic* fonts are either new proposals from KRACE or existing techniques customized to meet KRACE's purpose.

Fuzzing loop. Figure 3.12 shows the fuzzing evolution algorithm in KRACE. Fuzzing starts with producing a new program by merging two existing seeds. The seed selection criterion used in KRACE so far is simply frequency count, *i.e.*, less used seeds receive priority. We expect more advanced seed selection algorithms to be developed later. After merging, each program goes through several extension loops on which the program structure is altered with syscalls added and deleted. Each structurally changed program will further go through several modification loops in which the syscall arguments and distribution among the threads are mutated. Finally, each modified program runs repeatedly for several times, each with a different delay schedule, to probe for alias coverage.

Three parameters tunes the behaviors of the seed evolution loop: namely `ext_limit`, `mod_limit`, and `rep_limit` as shown in Figure 3.12. In KRACE, they take the values of 10, 10, and 5 respectively. That is,

```
def fuzzing_loop(ext_limit, mod_limit, rep_limit):
    while True:
        program = merge_seeds(select_seed_pair())

        ext_stall = 0
        while ext_stall < ext_limit:
            ext_stall++
            [50%] program.add_syscall()
            [50%] program.del_syscall()

            mod_stall = 0
            while mod_stall < mod_limit:
                mod_stall++
                [80%] program.mutate()
                [20%] program.shuffle()

                rep_stall = 0
                while rep_stall < rep_limit:
                    rep_stall++
                    delay = randomize_delay()
                    cov, log = run(program, delay)

                    if not cov.empty():
                        rep_stall = mod_stall = ext_stall = 0
                        schedule_data_race_check(log)
                        prune_and_save_seed(program)
```

Figure 3.12: The seed evolution process (a.k.a the fuzzing loop) in KRACE

- if any new coverage, either branch or alias, is observed in 5 consecutive runs, KRACE will continue to run the same multi-threaded seed for 5 more times but with a new delay schedule each time;
- if no new coverage is observed for 5 consecutive runs, KRACE starts to mutate the syscall arguments in the multi-threaded trace or shuffle the syscalls;
- if no new coverage is observed for 50 consecutive runs, KRACE starts to alter the input structure by adding or deleting the syscalls in the multi-threaded traces;
- if no new coverage is observed for 500 consecutive runs, KRACE starts to merge two seeds for a new seed.

In general, we give preference to alias coverage exploration over growing the multi-threaded syscall sequences, as we prefer to explore the concurrency domain as much as possible when the number of syscalls executed is small, making it easier for kernel developers to debug a reported data race.

Offline checking. Data race checking is conducted offline, *i.e.*, only when new coverage, either branch or alias, is found. The reason is that data race checking is slow (several minutes) and significantly hinders the fast fuzzing experience (which only requires a few seconds to finish one execution). As a result, we allow the fuzzers to quickly expand coverage and only dump execution logs without checking them. A few background threads check the execution logs for data races whenever they have free capacity. The checking progress has difficulty keeping up with seed generation in the beginning but will gradually catch up, especially when the coverage is toward saturation.

3.4.2 Benign vs harmful data races

An unexpected problem we encountered when reporting the data races found by KRACE is on differentiating benign and harmful data races. Despite the common belief that being data-race free is one of the coding practices in the kernel, benign data races are not totally uncommon. One major category is statistics accounting, such as `__part_stat_add` in the block layer. These statistics are meant for information and hints only and do not provide any accuracy guarantees. Another example is the reading and writing of different bits in the same 2-, 4-, 8-byte variable, especially bit-flags such as `inode->i_flag` or flags in file system control structures like `fs_info`.

Based on our experience, checking whether a data race is benign or harmful is often time consuming, as it requires careful analysis of the code and documentation to infer developers' intentions. In the worst cases, it may require consulting the file system developers, who may not even agree among themselves. One possibility to confirm a harmful data race is to keep the system running until the data race causes any visible effects such as violating assertions or memory errors. However, this is not always feasible, as shown in the case in Figure 2.1. It might need thousands of file operations running in parallel to trigger an integer overflow. By then, debugging such an execution trace will be another problem.

To avoid reporting benign data races to developers, KRACE uses several simple heuristics to filter the reports. In particular, a data race is mostly benign if:

- the race involves variables that have `stat` in their names or occurs within functions for statistics accounting;
- the race involves reading and writing to different bits of the same variable;
- the race involves kernel functions that can tolerate being racy, *e.g.*, `list_empty_careful`.

Unfortunately, these heuristics typically offer limited help for the more complicated cases.

3.4.3 The aging OS problem

When fuzzing file systems, most generic OS fuzzers do not reload a fresh copy of the kernel instance or file system image [32, 31, 38] for a new fuzzing session. Instead, they directly issue the syscall sequence on the old kernel state. The intention is to remove the overhead of kernel booting, as a VM emulator might take seconds to load and boot the kernel. However, this also means that any bugs found in this approach might come from the accumulated effects of hundreds or even thousands of prior runs, making them extremely difficult to debug and confirm by kernel developers, as is evident in the case when many bugs found by Syzkaller cannot be confirmed [98].

The aging OS problem is already difficult for fuzzing in the sequential domain, and bringing in the concurrency dimension further complicates the story, especially for KRACE, as the lengthy thread interleaving traces are not only difficult to debug but also renders analysis impossible. Slicing the execution traces does not seem feasible either, as cutting the trace at the wrong points means losing the locking and happens-before context, ultimately leading to false alarms. As a result, KRACE is forced to use a clean-slate execution for every fuzzing run, *i.e.*, a fresh kernel and a clean file system image.

The aging OS problem is also reported by Janus [9], which uses a library OS—LKL [99]—to enable quick reloading. But unfortunately, LKL does not support the symmetrical multi-processing (SMP) architecture, which is the prerequisite for multi-threading (*e.g.*, without SMP, all `spin_locks` becomes no-ops). As a result, LKL is mostly suitable for sequential fuzzing, not for concurrency fuzzing.

Table 3.2: Implementation complexity of KRACE in terms of LoC measurement of the major components shown in Figure 3.11.

Component	LoC	Languange
Compile-time preparation		
Kernel annotations	5,653	C
LLVM instrumentation pass	1,977	C++
KRACE kernel runtime library	1,749	C
Fuzzing loop		
Seed evolution (including syscall spec.)	9,394	Python
QEMU-based fuzzing executor	5,878	Python
Initramfs and the `init` program	2,527	Python
Data race checker	6,883	Python
Debugging tools and utilities	1,096	Python

3.5 Implementation

KRACE's code base is divided into two parts: 1) compile-time preparation, including annotations to the kernel source code (in the form of kernel patches), an LLVM instrumentation pass, and the KRACE library compiled into the kernel that provides coverage tracking and logging at runtime; and 2) a VM-based fuzzing loop that evolves test cases, executes them in QEMU VMs, and checks for data races. The complexity of each component is described in Table 3.2 and an overview of the runtime executor is shown in Figure 3.13.

Runtime executor. The most challenging part of KRACE's implementation is to establish information-sharing channels between the host and VM-based fuzzing instances for seed injection, coverage tracking, and feedback collection. KRACE uses private memory mapping (PCI memory bar), public memory mapping (`ivshmem`), and the `9p` file sharing protocols for this purpose, as shown in Figure 3.13.

Kernel building. Building the Linux kernel with LLVM is straightforward since kernel v5.3 and LLVM 9.0. In addition, to get the smallest possible boot time, we opt for a minimal kernel build with only necessary components enabled, including the block layer, loopback device, and all other related drivers to support and accelerate execution in QEMU and

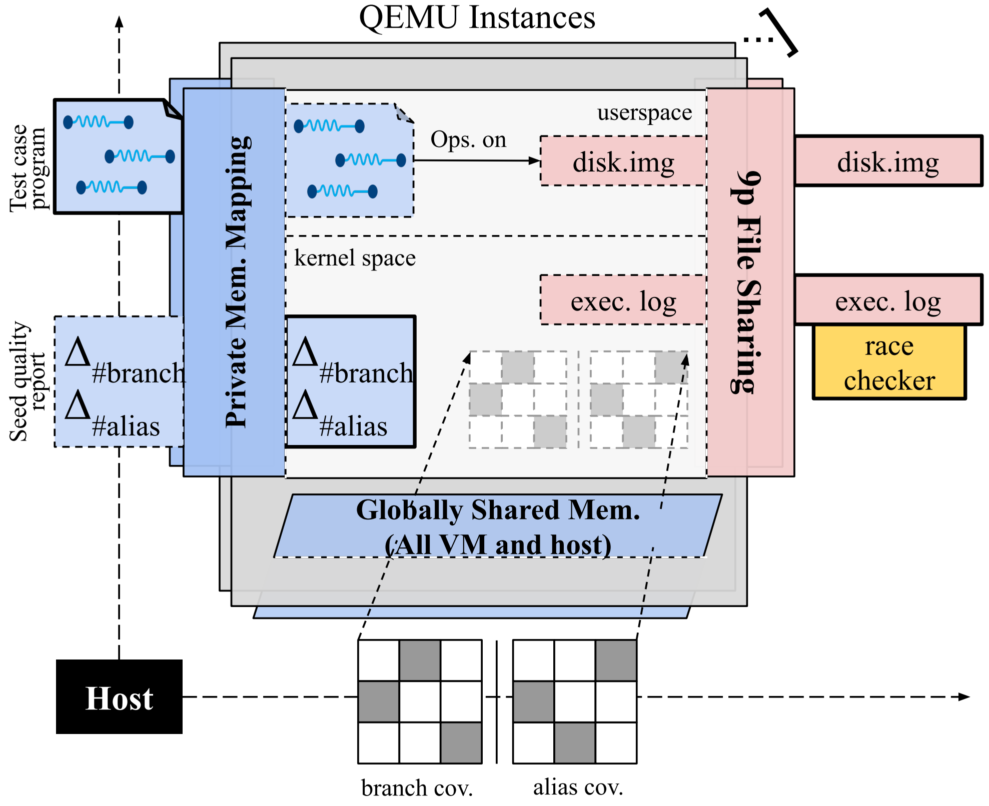


Figure 3.13: Implementation of the QEMU VM-based fuzzing executor in KRACE. The VM instance and the host have three communication channels: 1) private memory mapping, which contains the test case program to be executed by the VM and the seed quality report generated by KRACE runtime; 2) globally shared memory mapping, which contains the coverage bitmaps globally available to the host and all VM instances; 3) file sharing under the `9p` protocol for sharing of large files, including the file system image and the execution log.

KVM. File systems are built as modules, not built-in, and these modules will be loaded by our fuzzing agent (*i.e.*, the `init` program) such that we could track the modules in full, including the thread they fork on loading and their synchronization orders.

Initramfs. Again, to shorten the execution time, KRACE does not rely on full-blown OSes, not even tools like `busybox`, as they may interfere with the file system under testing. Instead, the `init` program in KRACE is the fuzzing agent that takes the multi-threaded seed and interprets it. In particular, the `init` 1) starts tracing, 2) loads file system modules, 3) mounts the file system image, 4) interprets the program, 5) unmounts the file system image, 6) unloads the modules, and 7) stops tracing.

Coverage tracking. Coverage tracking is handled by the instrumented code which are essentially stub calls, *e.g.*, `on_basic_block_enter`, `on_memory_read`, etc., into the KRACE runtime library. KRACE directly updates the coverage bitmaps maintained in the host memory regions that are globally visible to all VM instances (and their threads). Effectively, each update is a `test_and_set_bit` operation while the QEMU `ivshmem` protocol ensures atomicity.

Execution log. An execution log is simply an array of

```
[<event-type>, <thread-id>, <arg1>, <arg2>, ...]
```

filled by the KRACE runtime library and consumed by the data race checker for data race detection as well as reporting purposes such as call trace reconstruction.

3.6 Evaluation

In this section, we evaluate KRACE as a whole as well as per each component. In particular, we show the overall effectiveness of KRACE by listing previously unknown data races found (subsection 3.6.1); provide a comprehensive view of KRACE's performance characteristics, *e.g.*, speed, scalability, etc., as a file system fuzzer (subsection 3.6.2); justify major design decisions with controlled experiments (subsection 3.6.3); and compare KRACE against recent OS and data race fuzzers (subsection 3.6.4).

Experiment setup. We evaluate KRACE on a two-socket, 24-core machine running Fedora 29 with Intel Xeon E5-2687W (3.0GHz) and 256GB memory. All performance evaluations are done on Linux v5.4-rc5, although the main fuzzer runs intermittently across versions from v5.3. We build the kernel core with minimal components but enable as many features as possible for the `btrfs` and `ext4` file system modules. For all evaluations, the fuzzing starts with an empty file system image created from the `mkfs.*` utilities. We run 24 VM instances in parallel for fuzzing and each VM runs a three-thread seed.

Table 3.3: List of data races found and reported by KRACE so far. Status of benign* means that it is a benign race according to the execution paths we submitted, but the kernel developers suspect that there might be other paths leading to potentially harmful cases.

ID	FS	Racing access	Status
1	btrfs	heap struct: `cur_trans->state`	pending
2	btrfs	heap struct: `cur_trans->aborted`	harmful
3	btrfs	heap struct: `delayed_rsv->full`	harmful
4	btrfs	heap struct: `sb->s_flags`	benign
5	btrfs	global variable: `buffers`	harmful
6	btrfs	heap struct: `inode->i_mode`	benign
7	btrfs	heap struct: `inode->i_atime`	harmful
8	btrfs	heap struct: `BTRFS_I(inode)->disk_i_size`	harmful
9	btrfs	heap struct: `root->last_log_commit`	harmful
10	btrfs	heap struct: `free_space_ctl->free_space`	benign
11	btrfs	heap struct: `cache->item.used`	harmful
12	ext4	heap struct: `inode->i_mtime`	benign
13	ext4	heap struct: `inode->i_state`	benign
14	ext4	heap struct: `ext4_dir_entry_2->inode`	benign
15	ext4	heap array: `ei->i_data[block]`	harmful
16	VFS	heap string: `name` in `link_path_walk`	pending
17	VFS	heap struct: `inode->i_state`	benign
18	VFS	heap struct: `inode->i_wb_list`	benign
19	VFS	heap struct: `inode->i_flag`	benign
20	VFS	heap struct: `inode->i_opflags`	benign
21	VFS	heap struct: `file->f_mode`	benign*
22	VFS	heap struct: `file->f_pos`	pending
23	VFS	heap struct: `file->f_ra.ra_pages`	harmful

3.6.1 Data races in popular file systems

Across intermittent fuzzing runs on two popular kernel file systems (`btrfs` and `ext4`) during two months, KRACE found and reported 23 new data races, of which nine have been confirmed to be harmful, 11 are benign, and the rest of them are still under investigation, as listed in Table 3.3. Note that besides bugs in concrete file systems, KRACE also finds data races in the virtual file system (VFS) layer, which might affect all file systems in the kernel.

Consequence. Based on our preliminary investigation, only one bug (#5) is likely to cause immediate effects (null-pointer dereference) when triggered. Others are likely to cause

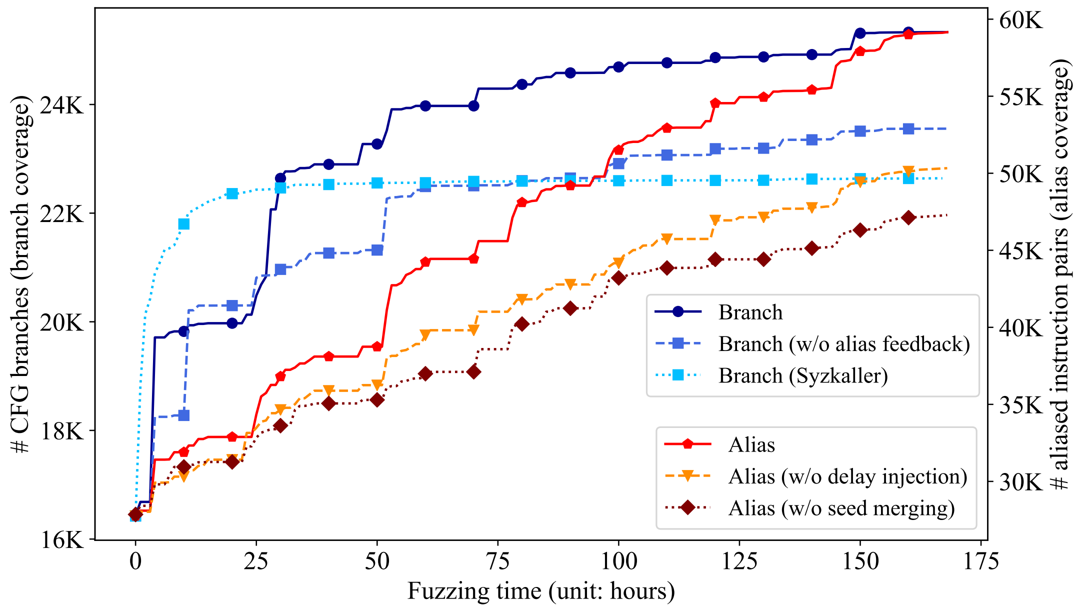


Figure 3.14: Evaluation of the coverage growth of KRACE when fuzzing the `btrfs` file system for a week (168 hours) with various settings.

performance degradation or specification violations, but we do not see a simple path toward memory errors. This also means that relying on bug signals such as KASan reports or kernel panics might not be sufficient to find data races.

3.6.2 Fuzzing characteristics

Coverage growth. The growth patterns for both branch and alias coverage are plotted in Figure 3.14 (for `btrfs`) and Figure 3.15 (for `ext4`). There are several interesting observations:

Alias coverage size. Although branch coverage for the two file systems grow into roughly the same level (25K vs 20K), compared with `ext4`, `btrfs` has a significantly larger alias coverage bitmap, (60K vs 9K). Given that the number of user threads is the same (3 threads), the difference is caused by the level of concurrency inherent in `btrfs` and `ext4` design. As shown in Figure 3.5, `btrfs` uses at least 22 background threads and each thread may additionally fork more helper threads, while the only background thread for `ext4` is the

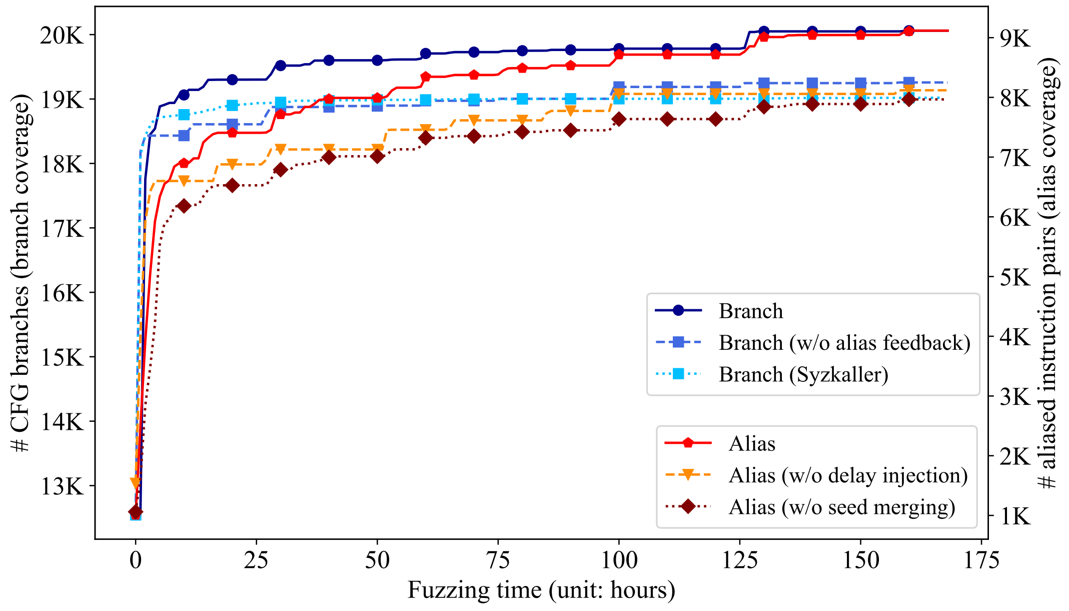


Figure 3.15: Evaluation of the coverage growth of KRACE when fuzzing the `ext4` file system for a week (168 hours) with various settings.

`jbd2` journaling thread. In other words, `btrfs` is inherently more concurrent than `ext4`, and dividing works among more threads naturally leads to more alias pairs. The similar logic also applies to why alias coverage saturates much faster in `ext4`, the less concurrent file system.

Growth synchronization. In general, the two coverage metrics grow in synchronization. It is expected that progresses in the branch coverage will yield new alias coverage too because new code paths mean new memory accessing instructions and hence, new alias pairs. However, it is the other direction that matters more: branch coverage saturates but alias coverage keeps growing, *e.g.*, starting from hour 75 in the `btrfs` case or hour 25 in the `ext4` case. In other words, KRACE keeps finding new execution states (thread interleavings) that would otherwise be missed if only branch coverage is tracked.

Instrumentation overhead. The code instrumentation from KRACE is heavy, and we expect it to cause significant overhead in execution. To show this, we present the aggregated statistics on the execution time for seeds bearing different numbers of syscalls. For compari-

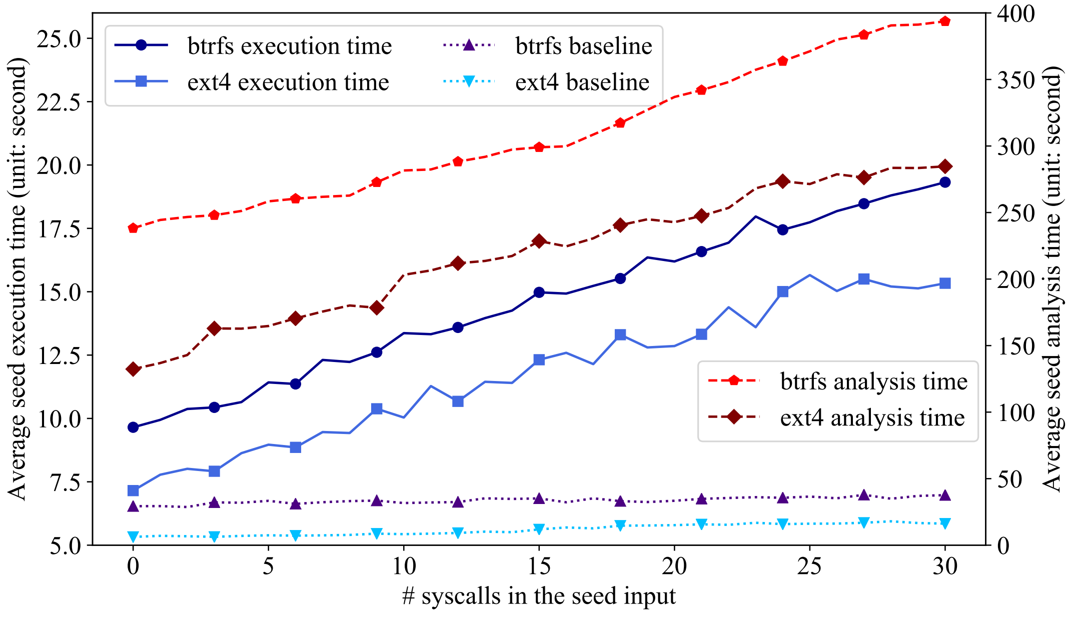


Figure 3.16: Evaluation of seed execution and analysis time in KRACE with a varying number of syscalls in the seed

son, we also run these seeds on a bare-metal kernel built without KRACE instrumentation. The results are plotted in Figure 3.16. In summary, in the zero-syscall case, *i.e.*, by merely loading (file system module) → mounting (image) → unmounting → unloading, KRACE already incurs 47.6% and 34.3% overhead, and the more syscalls KRACE executes, the more overhead it accumulates.

The overhead mainly comes from memory access instrumentation, as every memory access is now turned into a function call where atomic operations are performed and synchronized, not only with respect to all other threads on the VM, but also against all threads across all VMs, as the thread is updating the global bitmap on the host directly (implicitly handled by the QEMU `ivshmem` module). As a result, further optimizations are possible. For example, a VM instance may accumulate coverage locally and update the global bitmap in batches instead of on every memory access.

It is, however, debatable whether the overhead is detrimental to KRACE since lower overhead simply means that the coverage growth will converge and saturates faster. In our opinion, we consider the overhead caused by tracking more coverage (including alias coverage) as a trade-off between execution speed and seed quality. A fuzzer with fast executions may waste resources in non-interesting test cases, while a fuzzer with slow executions but finer-grained tracking might eventually have higher chances to explore more states.

Data race checking cost. Another limiting factor for KRACE is the time needed to analyze the execution logs for data race detection, which also depends on the length of the execution trace. The trend is also plotted in Figure 3.16. In summary, the analysis time ranges from 4-7 minutes (0-30 syscalls per seed) for `btrfs` and 2-6 minutes for `ext4`. Such a time cost is obviously not feasible for online checking (even after optimization) but can be tolerated for offline checking, *i.e.*, KRACE schedules a data race check only when a seed is discovered. This strategy works especially when fuzzing saturates, as the bottleneck for making further progress then becomes finding new execution states instead of checking the trace. Based on our experience, running four checker processes alongside 24 fuzzing VM instances is more than sufficient to catch up to the progress within 96 hours in both cases.

3.6.3 Component evaluations

Coverage effectiveness. Although the two coverage metrics represent different aspects of program execution, we are also curious whether tracking explorations in the concurrency dimension may help in finding new code paths (represented by branch coverage). To check this, we disabled the alias coverage feedback and let KRACE explore the states mimicking the feedback loop of existing OS and file system fuzzers. The results (Figure 3.14 and Figure 3.15) show that exploring the concurrency domain also helps to find new code coverage. Most notably, without alias coverage feedback, branch coverage grows much faster at the beginning, because it does not spend fuzzing effort on exploring the thread interleavings, but saturates at a lower number (7.2% and 4.0% less). Moreover, if just counting the new branches explored (besides the branches in the initial seed), the coverage reduces by 20.4% and 10.7%, respectively. The more concurrent the file system is, the more branch coverage will be explored by enabling alias coverage feedback. This is not surprising, as certain code paths exist to handle contention in the system, such as the paths executed when `try_lock` fails or when sequence lock retries. Exploring in the concurrency dimension helps to reveal these paths and boost the branch coverage.

Delay injection effectiveness. To test whether injecting delays helps in exploration in the concurrency dimension, we disabled delay injection in this fuzzing experiment, and the alias coverage growth is shown in Figure 3.14 and Figure 3.15. With delay injection disabled, KRACE found 28.7% and 12.3% less alias coverage in `btrfs` and `ext4`, respectively. This shows that delay injection is important in finding more alias coverage. Especially, when the branch coverage saturates, delay injection becomes the leading force in finding alias coverage, as shown by the enlarging gap between the growth. The more concurrent the file system is, the more important delay injection becomes.

Seed merging effectiveness. To test whether reusing the seed helps in exploration in the concurrency dimension, we disabled seed merging in this fuzzing experiment, *i.e.*, KRACE only adds, deletes, and mutates syscalls but never reuses the found seeds. The alias coverage growth is shown in Figure 3.14 and Figure 3.15. With seed merging disabled, KRACE found 37.7% and 14.2% less alias coverage in `btrfs` and `ext4`, respectively. This experiment shows that reusing the seed is important in quickly expanding the coverage. More importantly, preserving the semantics among the syscalls and interleaving the seeds help find more alias coverage.

Components in the data race checker. To show that it is important to have both happens-before and lockset analysis (and their sub-components) in the data race checker, we sampled a simple fuzzing run: load `btrfs` module, mount an empty image, execute two syscalls $\times$ three threads, unmount the image, and unload the `btrfs` module. The following shows the filtering effects of each component in the data race checker:

- data race candidates: 35,658

+ after lockset analysis on pessimistic locks: 13,347
+ after lockset analysis on optimistic locks: 8,903
+ after tracking fork-style happen-before relation: 6,275
+ after tracking join-style happen-before relation: 3,509
+ after handling publisher-subscriber model: 103
+ after handling ad-hoc schemes: 7 (all benign races)

3.6.4 Comparison with related fuzzers

Execution speed vs coverage. In terms of efficiency, KRACE is not comparable to other OS and file system fuzzers, as one execution takes at least seven seconds in KRACE, while the number can be as low as 10 milliseconds for libOS-based fuzzers [9, 17] or never-refreshing VM-based fuzzers like Syzkaller. However, the effectiveness of a fuzzer is not solely decided by fuzzing speed. A more important metric is the coverage size, especially when saturated. Intuitively, if the saturated coverage is low, being fast in execution only implies that the coverage will converge faster and mostly stall afterward.

On the metric of saturated coverage, KRACE outperforms Syzkaller for both `btrfs` and `ext4` by 12.3% and 5.5%, respectively, as shown in Figure 3.14 and Figure 3.15. Even without the alias coverage feedback, the branch coverage from KRACE still outperforms Syzkaller, showing the effectiveness of KRACE's seed evolution strategies, especially the merging strategy for multi-threaded seeds, which is currently not available in Syzkaller. In fact, KRACE is able to catch up to the branch coverage progress with Syzkaller within 30 hours and eight hours for `btrfs` and `ext4`, respectively.

Data race detection. Razzer [39] reports four data races in file systems and we find the patches for two of them, both in the VFS layer. To check that KRACE may detect these cases, we manually revert the patches in the kernel and confirm that both cases are found. We would like to do the same for SKI [18], but the data races found by SKI are too old (in 3.13 kernels) and locating and reverting the patches is not easy.

3.7 Discussion and Limitations

Deterministic replay. Being able to replay an execution deterministically is extremely helpful for debugging and also opens the door for advanced data race triaging techniques such as controlled re-interleaving of thread executions. Unfortunately, we are sorry to report that even with a totally linearized trace of basic block enter/exit, memory accesses, lock acquisition/releases, and kernel synchronization API calls, KRACE is unable to deterministically replay an execution end-to-end. Part of the reason is the missing instrumentation in other kernel components, including the kernel core (including the task and IO scheduler), memory management, device drivers (except the block device), and most of the library routines. We expect that deterministic replay may be possible if we instrument all kernel components but at the expense of huge execution footprints (*e.g.*, GB-level logs) as well as significant performance drops. We are unaware of a system that permits deterministic replay of over 60 kernel threads, but we are eager to integrate if possible.

Debuggability. To partially compensate for not being able to replay a found data race deterministically, KRACE tries to generate a comprehensive report for each data race, including 1) the conflicting lines in source code, 2) the full call stack for each thread, and 3) the callback graph. Since each instruction is labeled with a compile-time random number, KRACE is able to pinpoint the conflicting lines in the source code when a data race occurs. Further coupled with the basic block branching information, KRACE is able to recover the full call trace, up to the syscall entry point or the thread creation point, for all involving threads during the race condition. The report may also involve the callback graph derived from the happens-before analysis, to further assist the developers with the origin of the threads. In fact, kernel developers have never asked for a deterministic replay of the trace and are able to judge whether the race is harmful or benign based on the information provided.

Missing bugs. Offlining the data race checker means that KRACE might miss data race bugs. As discussed in subsection 3.1.2, alias coverage is just an approximation of state exploration progress in the concurrency dimension, and there might be new program states explored at runtime but that do not show up as new coverage, *i.e.*, meaningful interleavings missed by alias coverage. KRACE forgoes the opportunities to check data races in those cases and is a trade-off made in favor of expanding the coverage with efficiency.

CHAPTER 4
DEADLINE: PRECISE DETECTION OF KERNEL DOUBLE-FETCH BUGS

4.1 Double-fetch Bugs: a Formal Definition

As discussed in subsection 2.1.2, relying on empirical code patterns for *double-fetch bug* detection is imprecise and could result in a lot of manual effort to verify that a *multi-read* is indeed a *double-fetch bug*. Instead, DEADLINE labels an execution path as a *double-fetch bug* when the following four conditions are met:

1. There are at least two reads from userspace memory, i.e., it must be a *multi-read* while a userspace fetch can be identified by transfer functions like `copy_from_user`.
2. The two fetches must cover an overlapped memory region in the userspace. If this condition is met, we call the *multi-read* an *overlapped-fetch*.
3. A *relation* must exist based on the overlapped regions between the two fetches. We consider both control and data dependence as relations.
4. DEADLINE cannot prove that the relation established still holds after the second fetch. In other words, a user process can do a race condition to change the content in the overlapped region to destroy the relation.

Conditions 1) and 2) are straightforward to understand. For condition 3), if the execution path can be deviated based on the values from the first fetch, it implies an assumption about these values, and this assumption should be honored by the second fetch. A typical example is shown in Figure 2.3, whereby after the first fetch, the control flow is deviated if `header.version != TLS_1_2_VERSION`, i.e., the second fetch can never happen. The fact that line 13 can be reached already implies that `header.version == TLS_1_2_VERSION`, which is not re-checked after the second fetch and this makes it a *double-fetch bug*.

For data dependence, consider the bug shown in Figure 2.2, where the value `khdr.iocnum` is used to look up the correct adapter, `iocp`, to handle the request. The fact that line 18 (the second fetch) can be reached implies that an adapter is already found and a `mutex` is already held. However, in line 22, the adapter is looked-up again (with `kfwdl.iocnum`), but this time, an adapter different from `iocp` can be found if the `iocnum` is changed, leading to a request performed without the intended adapter whose `mutex` is held.

It is also possible that both control and data dependence exist. This typically happens when a variable representing total message size is fetched in, sanity-checked, and later used to do the second fetch, as shown in Figure 4.1a. The variable `size` must be within a reasonable range, and `attr->size` should hold the effective size of the `attr` buffer. However, after the second fetch, both relations might not hold anymore.

For condition 4), if the relation established in condition 3) is control dependence only, we need to prove that the same set of constraints still holds for the values copied in after the second fetch. In the example of Figure 2.3, we should check that `full->version == TLS_1_2_VERSION` still holds. On the other hand, if a data dependence is established, re-checking the conditions is not sufficient and a full equality proof is needed. In the case of Figure 4.1a, checking that `PERF_ATTR_SIZE_VER0 <= attr->size <= PAGE_SIZE` does not reflect the relation that `attr->size` holds the effective size of `attr`. The correct way is to prove that `attr->size == size` in all cases.

Put the above description in formal terms:

Fetch. We use a pair (A, S) to denote a *fetch*, where A represents the starting address of the *fetch* and S represents the size of the memory (in bytes) copied into kernel.

Overlapped-fetch. Two fetches, (A_0, S_0) and (A_1, S_1), are considered to have an overlapped region if and only if:

$$A_0 <= A_1 < A_0 + S_0 \quad or \quad A_1 <= A_0 < A_1 + S_1$$

Correspondingly, we use a pair (A_{01}, S_{01}) to denote the overlapped memory region for the two fetches and a triple $(A_{01}, S_{01}, i = [0, 1])$ to denote the memory copied in during the first or second fetch.

Control dependence. A variable V is considered to be control dependent if $V \in (A_{01}, S_{01}, 0)$ and V is subject to a set of constraints in order for the second fetch to happen. The set of constraints V must satisfy is denoted as $[V_c]$. To prove that a *double-fetch bug* cannot exist in this case, we have to prove that V' constructed from $(A_{01}, S_{01}, 1)$ must also satisfy $[V_c]$.

Data dependence. A variable V is considered to be data dependent if $V \in (A_{01}, S_{01}, 0)$ and V is consumed, such as being assigned to other variables, involved in calculations, or passed to function calls. To prove that a *double-fetch bug* cannot exist in this case, we have to prove that V' constructed from $(A_{01}, S_{01}, 1)$ must satisfy $V' == V$.

4.2 DEADLINE Overview

The formal modeling of *double-fetch bugs* inspired us to use symbolic checking for *double-fetch bug* detection. Compared with actually executing the code with concrete inputs (like Bochspwn [43]), symbolic execution gives us the power of generality, i.e., *solving* for a case that can meet certain conditions or *proving* that these conditions can never be satisfied. This gives DEADLINE the precision of detection and also alleviates the manual effort. Furthermore, symbolic execution, being a static analysis technique, is not limited by the availability of hardware or machine configurations and can theoretically be applied against all the drivers, file systems, and peripheral modules in the kernel source tree. It also enables DEADLINE to start path exploration and checking from virtually any point, in a manner similar to UC-KLEE [100], instead of from fixed entry points like syscall entry or kernel boot.

However, before rushing into symbolic checking, DEADLINE needs to collect as many *multi-reads* as possible since every *double-fetch bug* must be a *multi-read*. Furthermore, for each *multi-read*, DEADLINE constructs the execution paths for the symbolic execution to follow along. DEADLINE achieves this by first compiling kernel source code into the LLVM intermediate representation (IR) and statically analyzing the IR to identify *multi-reads* and prune associated execution paths.

We choose to work with LLVM IR instead of the C source code for several reasons: 1) LLVM IR preserves most of the information needed by DEADLINE, such as type information, function names and arguments, etc. The only information loss is the `__user` mark, which can be easily added back to the IR by adding the mark to LLVM metadata. The `__user` mark is used to differentiate userspace and kernel memory objects, which will be illustrated in detail in subsection 4.4.3. 2) LLVM IR is in the single static assignment (SSA) form, which closely mimics the logic for symbolic execution, i.e., assigning a symbolic value to each definition of a variable. Working with LLVM IR also enables the reuse of many LLVM analysis passes such as call-graph construction, function inlining, etc.

Overall procedure. In short, DEADLINE's detection procedure consists of two steps:

1. Scan the kernel and collect as many *multi-reads* as possible with their execution paths.
2. Check along each execution path to see if a *multi-read* turns into a *double-fetch bug*.

The high-level procedure for DEADLINE is illustrated in algorithm 1. The process starts by scanning the kernel and collecting all fetches (line 2). This is an easy step, as each userspace fetch is clearly marked by a call to a transfer function (*e.g.*, `get_user`). Then, for each fetch found, DEADLINE scans backward and forward along the control flow graph for other fetches (line 4), and if other fetches are found, marks them as fetch pairs, each constitutes a *multi-read*. Afterward, DEADLINE builds all possible execution paths that run through both fetches for each fetch pair (line 6). We elaborate these two steps in section 4.3. Finally, for each execution path constructed, DEADLINE invokes its symbolic execution engine and checks whether the *multi-read* is a *double-fetch bug* based on the formal definitions. This step is depicted in detail in section 4.4.

Algorithm 1: High-level procedure for *double-fetch bug* detection

```
In  : $Kernel$ - The kernel to be checked
Out : $Bugs$ - The set of double-fetch bugs found
1  $Bugs \leftarrow \emptyset$
2  $Set_f \leftarrow$ Collect_Fetches($Kernel$)
3  for $F \in Set_f$ do
4      $Set_{mr} \leftarrow$ Collect_Multi_Reads($F$)
5      for $< F_0, F_1, Fn > \in Set_{mr}$ do
6          $Paths \leftarrow$ Construct_Execution_Paths($F_0$, $F_1$, $Fn$)
7          for $P \in Paths$ do
8              if Symbolic_Checking($P$, $F_0$, $F_1$) == UNSAFE then
9                  $Bugs$.add($< F_0, F_1 >$)
10             end
11         end
12     end
13 end
```

4.3 Finding Multi-reads

Finding *multi-reads* is the first step for *double-fetch bug* detection. Prior works either used empirical rules [44] or relied on dynamic memory access patterns [43] to find *multi-reads*, both of which could be problematic (e.g., assuming *multi-reads* are intra-procedural). To inherently improve the finding of *multi-reads*, DEADLINE instead employs static and symbolic program analyses to systematically find *multi-reads* against the whole kernel codebase.

4.3.1 Fetch pairs collection

In this step, the goal for DEADLINE is to statically enumerate all *multi-reads* that could possibly occur. In particular, DEADLINE tries to identify all the fetch pairs that can be reached at least statically, i.e., there exists a reachable path in the control flow graph (CFG) between the two fetches (i.e., a fetch pair).

One approach is to 1) identify all fetches in the kernel, i.e., calls to transfer functions; 2) construct a complete, inter-procedural CFG for the whole kernel; and 3) perform pair-wise reachability tests for each pair of fetches. Although 1) is easy, given the scale and complexity of kernel software, both 2) and 3) are hard if not impossible in practice. Therefore, DEADLINE chooses to find fetch pairs in a bottom-up manner, as described in algorithm 2. In short, starting at each fetch, within the function it resides in, DEADLINE scans through both the reaching and reachable instructions for this fetch and among those instructions, either marks that we have found a fetch pair (line 6, 15) or inline the function containing a fetch and re-executes the search (line 9, 18).

Algorithm 2: Collect_Multi_Reads(F)

```
   In  : $F$ - A fetch, i.e., a call to a transfer function
   Out : $R$ - A set of triples $< F_0, F_1, Fn >$ representing *multi-reads*
1  $Fn \leftarrow$ Function that contains $F$
2  $R \leftarrow \emptyset$
3  $Set_{up} \leftarrow$ Get_Upstream_Instructions($Fn$, $F$)
4  for $I \in Set_{up}$ do
5      if $I$ is a fetch then
6          $R$.add($< I, F, Fn >$)
7      end
8      if $I$ is a call to a function that contains a fetch then
9          Inline $I$, redo the algorithm
10     end
11 end
12 $Set_{dn} \leftarrow$ Get_Downstream_Instructions($Fn$, $F$)
13 for $I \in Set_{dn}$ do
14     if $I$ is a fetch then
15         $R$.add($< F, I, Fn >$)
16     end
17     if $I$ is a call to a function that contains a fetch then
18         Inline $I$, redo the algorithm
19     end
20 end
```

Note that, in addition to the two fetches, the enclosing function Fn is also attached to the pair, and we use this triple to denote a *multi-read* in DEADLINE. Conceptually, this Fn is the deepest function in the global call graph (if it can ever be constructed) that encloses both fetches, and later the execution paths will be constructed within this Fn. This alleviates DEADLINE in constructing execution paths from fixed entry and exit points such as syscall enter and syscall return, which are usually very lengthy with many irrelevant instructions to the forming of a *double-fetch bug*.

Indirect calls. One special case in this process is an indirect call, which is often used in kernel to simulate polymorphic behaviors. DEADLINE does not attempt to resolve the actual targets of an indirect call (in fact, in many cases, they can only be resolved at runtime). Instead, DEADLINE conservatively identifies all potential targets of an indirect call. Specifically, DEADLINE first collects the address-taken functions and then employs the type-analysis-based approach [101, 102] to find the targets of indirect calls. That is, as long as the type of arguments of an address-taken function matches with the call-site of an indirect call, we assume it is a valid target of the indirect call.

4.3.2 Execution path construction

In this step, DEADLINE is given a triple $< F_0, F_1, Fn >$ which represents a *multi-read*, and the goal for DEADLINE is twofold:

1. to find all execution paths within the enclosing function (Fn) that connect both fetches (F_0 and F_1) and
2. to slice out the irrelevant instructions that have no impact on the fetches or are not affected by the fetches, for each execution path.

Both parts can be solved with standard program analysis techniques. The first part can be done by a simple CFG traversal within the function Fn, while the second part can be achieved by slicing the function CFG with the following criteria:

- An instruction is considered to *have an impact on a fetch* if the address or size of the fetch is either derived from it or constrained by it.
- An instruction is considered to *be affected by a fetch* if it is derived from the fetched-in value or it constrains the fetched-in value.

With these criteria, we preserve all the control and data dependence relations that we need to prove and thus to decide whether a *multi-read* is a *double-fetch bug*, as defined in section 4.1.

Linearize an execution path. One last step in the path construction is to linearize the paths into a sequence of IR instructions. For a path without loops, linearization is simply a concatenation of the basic blocks; however, for a path with loops, unrolling is required. DEADLINE decides to unroll a loop only once[1]. This imposes a limitation to DEADLINE: DEADLINE is unable to find *double-fetch bugs* caused by one fetch overlapping with itself when the loop is executed multiple times. In fact, in kernel, such *double-fetch bugs* can almost never happen, as fetches in loops are usually designed in an incremental manner, e.g., `copy_from_user(kbuf, ubuf, len); ubuf += len;`. In this case, the two fetches are always from non-overlapping memory regions and will never satisfy the condition for a *double-fetch bug*. On the other hand, unrolling the loop once does help DEADLINE find *double-fetch bugs* caused by two fetches across loops, as shown in Figure 2.4.

4.4 From Multi-reads to Double-fetch Bugs

Prior works [43, 44] rely on manual verification to check whether a *multi-read* turns into a *double-fetch bug*, which can be time consuming and error-prone. Instead, DEADLINE applies symbolic checking to automatically vet whether a *multi-read* is a *double-fetch bug* based on the formal definitions in section 4.1.

4.4.1 A running example

To help illustrate the concepts in this section, we provide a running example in Figure 4.1. It is a *double-fetch bug* found by DEADLINE in the `perf_copy_attr` function and has been patched in Linux kernel 4.13. In summary, the first fetch (line 8) copies in a 4-byte value

[1]If there are multiple paths inside the loop body, DEADLINE unrolls the loop multiple times, each covers one path.

`size`, which is later sanity checked (line 12, 13) and also used for the second fetch (line 17). However, after the second fetch, the overlapped region `attr->size` is not subject to any constraints until the end of the function. In this case, a user process could put a proper value, say `uattr->size = 128`, before the first fetch so that it will pass both sanity checks and later uses a race condition to change it, say `uattr->size = 0xFFFFFFFF`, which will be copied to `attr->size` and cause trouble if it is later used without caution (line 24). The memory access pattern is also visualized in Figure 4.1b, which clearly shows that the two fetches have an overlap of four bytes and the constraints on this overlapped region across different fetches are different.

Given that both control dependence (i.e., the sanity checks) and data dependence (i.e., `size` is used in the second fetch) are established between the two fetches, the correct way to check for a *double-fetch bug* is to try an equality proof (i.e., proving that `size == attr->size`), as explained in section 4.1. Since this cannot be proved, DEADLINE flags this *multi-read* as a *double-fetch bug*.

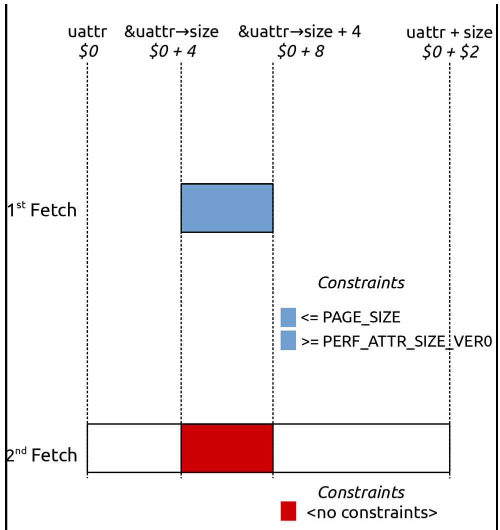
The symbolic execution procedure is shown in Figure 4.1c. Note that, for illustration purpose, we use $\$X$ to denote the symbolic value of the variable and $@X$ to denote the object in memory that is pointed to by $\$X$. If $\$X$ is not a pointer, $@X$ is `nil`. A memory object can be accessed by a triple $< i, j, L >$, which means a memory access from byte i to byte j. The label L can be either K or U, indicating whether this is a kernel or userspace access, and for userspace accesses, the labels can be $U0$, $U1$, etc. to denote that this is the first or second access to that memory object region.

Figure 4.2 presents a much more complicated example to illustrate DEADLINE's symbolic execution process. In particular, this examples shows two features of DEADLINE:

- *Loop unrolling*: each of the two `while` loops are unrolled once which is further reflected in the symbolic execution trace: line 9-16, 22-35 (4.2c).
- *Pointer resolving*: in DEADLINE's memory model, if DEADLINE can prove that two pointer values are the same, they should be pointing to the same object, which is

```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
      size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // second fetch
  if (copy_from_user(attr, uattr, size))
    return -EFAULT;

  ......
}
// Example: if attr->size is used later
// BUG: attr->size can be very large
memcpy(buf, attr, attr->size);
```

(a) C source code



(b) Memory access patterns

```
// init root SR
$0 = $PARM(0),     @0 = $UMEM(0) // uattr
$1 = $PARM(1),     @1 = $KMEM(0) // attr
---
// first fetch
fetch(F1) is {A = $0 + 4, S = 4}
$2 = @0(4, 7, U0),@2 = nil       // size
---
// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0
---
// second fetch
fetch(F2) is {A = $0, S = $2}
@1(0, $2 - 1, K) = @0(0, $2, U1)
---
// check fetch overlap
assert F2.A <= F1.A < F2.A + F2.S
    OR F1.A <= F2.A < F1.A + F1.S
// --> satisfiable with @0(4, 7, U)

// check double-fetch bug
prove @0(4, 7, U0) == @0(4, 7, U1)
// --> fail, no constraints on @0(4, 7, U1)
```

(c) Symbolic representation and checking

Figure 4.1: A *double-fetch bug* in `perf_copy_attr`, with illustration on how it fits the formal definition of *double-fetch bugs* (4.1b) and DEADLINE's symbolic engine can find it (4.1c).

reflected in line 20 (4.2c).

This example also shows that although developers tend to exercise precaution to prevent *double-fetch bugs*, for example, by placing the sanity checks at line 36 (4.2a), such checks might not be sufficient as shown in line 28 (4.2c).

4.4.2 Transforming IR to SR

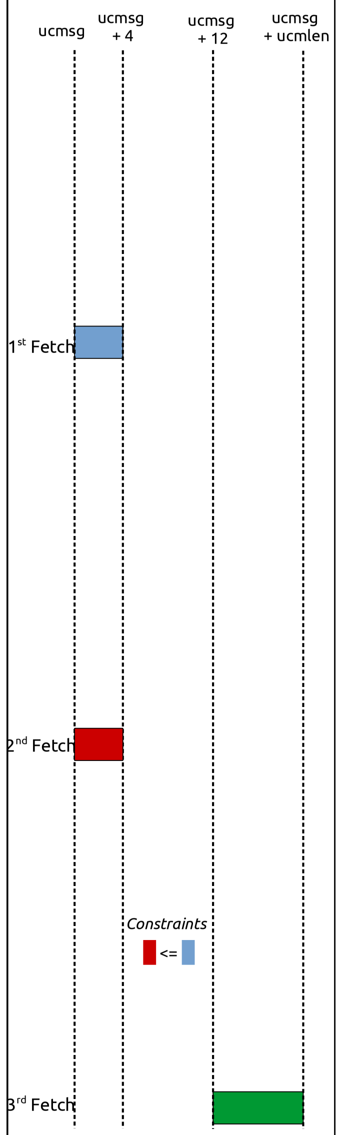
Transforming the LLVM IR to SR is the same as symbolically executing the LLVM instructions along the path. In particular, each variable has an SR while the instructions and function calls define how to derive these SRs and the constraints imposed. All the SRs are derived from a set of root SRs, which could be function arguments, global variables (both denoted as `PARM`), or two special types of objects, `KMEM` and `UMEM`, that represent memory blobs in kernel and userspace, respectively. Function arguments and global variables are considered roots because their values are not defined by any instructions along the execution path. Similarly, the initial contents in `KMEM` and `UMEM` are unknown, and therefore we also treat them as root SRs, although along the execution their contents can be defined through operations such as `memcpy`, `memset`, and `copy_from_user`.

Symbolic execution of the majority of LLVM instructions is straightforward. For example, to symbolically execute an `add` instruction `%3 = add i32 %2 16`, DEADLINE simply creates a new SR, `$3`, and sets it to `$3 = $2 + 16`. However, three types of instructions need special treatment: branch instructions, library functions/inline assemblies, and memory operations.

Branch instructions. As stated before, DEADLINE does not perform new path discovery during symbolic execution; instead, it only follows along a specific path (i.e., a sequence of IR instructions) prepared before the symbolic execution, as illustrated in detail in subsection 4.3.2. Therefore, whenever DEADLINE encounters a branch instruction, it looks ahead on the path, checks which branch is taken, and uses this information to infer the constraints that must be satisfied by taking that branch, i.e., whether the branch condition

```c
int cmsghdr_from_user_compat_to_kern
  (struct msghdr *kmsg, char *kbuf) {

  struct compat_cmsghdr __user *ucmsg;
  compat_size_t ucmlen;
  struct cmsghdr *kcmsg;
  __kernel_size_t kcmlen, tmp;

  // 1st loop: calculate message length
  kcmlen = 0;
  ucmsg = kmsg->msg_control;
  while (ucmsg != NULL) {
    // first batch of fetches
    if (get_user(ucmlen, &ucmsg->cmsg_len))
      return -EFAULT;

    tmp = ucmlen + sizeof(struct cmsghdr)
          - sizeof(struct compat_cmsghdr);

    kcmlen += tmp;
    ucmsg = (char *)ucmsg + ucmlen;
  }

  // 2nd loop: copy the whole message
  kcmsg = kbuf;
  ucmsg = kmsg->msg_control;
  while (ucmsg != NULL) {
    // secind batch of fetches
    if (get_user(ucmlen, &ucmsg->cmsg_len))
      return -EFAULT;

    tmp = ucmlen + sizeof(struct cmsghdr)
          - sizeof(struct compat_cmsghdr);

    // sanity check, but insufficient
    if (kbuf + kcmlen - (char *)kcmsg < tmp)
      return -EINVAL;

    // irrelevant fetch
    if (copy_from_user(
        (char *)kcmsg + sizeof(*kcmsg),
        (char *)ucmsg + sizeof(*ucmsg),
        (ucmlen - sizeof(*ucmsg))))
      return -EFAULT;

    kcmsg = (char *)kcmsg + tmp;
    ucmsg = (char *)ucmsg + ucmlen;
  }

  // BUG: the actual message length != kcmle
  kmsg->msg_controllen = kcmlen;
  return 0;
}
```

(a) C source code



(b) Memory access patterns

```
// init root SR
$0  = $PARM(0),              @0  = $KMEM(0)   // kmsg
$1  = $PARM(1),              @1  = $KMEM(1)   // kbuf
---
// prepare for the 1st batch of fetches
$2  = 0,                     @2  = nil        // kcmlen_0
$3  = @0(48, 55, K),         @3  = $UMEM(0)   // ucmsg_0
---
// unroll 1st loop
assert $2 != NULL
fetch(F1) is {A = $3 + 0, S = 4}
$4  = @3(0, 3, U0),          @4  = nil        // ucmlen_0
$5  = $4 - 12 + 16,          @5  = nil        // tmp_0
$6  = $2 + $5,               @6  = nil        // kcmlen_1
$7  = $3 + $4,               @7  = $UMEM(1)   // ucmsg_1
assert $7 == NULL (i.e., @7 = nil)            // exit loop
---
// prepare for the 2nd batch of fetches
$8  = $1                     @8  = $KMEM(1)   // kcmsg_0
$9  = @0(48, 55, K) == $3,   @9  = @3         // ucmsg_2
---
// unroll 2nd loop
assert $9 != NULL
fetch(F2) is {A = $9 + 0, S = 4}
$10 = @3(0, 3, U1),          @10 = nil        // ucmlen_1
$11 = $10 - 12 + 16,         @11 = nil        // tmp_1

assert $1 + $6 - $8 >= $11  --> @3(0, 3, U0) >= @3(0, 3, U1)

fetch(F3) is {A = $9 + 12, S = $10 - 12}
@8(12, $10 - 13, K) = @3(12, $10 - 13, U0)

$12 = $8 + $11,              @12 = $KMEM(2)   // kcmsg_1
$13 = $9 + $10,              @13 = $UMEM(3)   // ucmsg_3
assert $13 == NULL (i.e., @13 = nil)          // exit loop
---

// check fetch overlap
assert F2.A <= F1.A < F2.A + F2.S
   AND F1.A <= F2.A < F1.A + F1.S
// --> satisfiable with @3(0, 3, U)

assert F3.A <= F1.A < F3.A + F3.S
   AND F1.A <= F3.A < F1.A + F1.S
// --> unsatisfiable

assert F3.A <= F2.A < F3.A + F3.S
   AND F2.A <= F3.A < F2.A + F2.S
// --> unsatisfiable

// check double-fetch bug
prove @3(0, 3, U0) == @3(0, 3, U1)
// --> fail, as @3(0, 3, U0) >= @3(0, 3, U1)
```

(c) Symbolic representation and checking

Figure 4.2: A *double-fetch bug* in `cmsghdr_from_user_compat_to_kern`, with illustration on how it fits the formal definition of *double-fetch bugs* (4.2b) and DEADLINE's symbolic engine can find it (4.2c).

is true or false. After doing that, DEADLINE adds this constraint to its assertion set so that later when solving (or proving), it ensures that this constraint is met (or cannot be met). In the running example in Figure 4.1, line 10, 11 in Figure 4.1c illustrate this procedure. This is in contrast to traditional symbolic executors [47, 48, 49] which fork states and try to cover both branches upon encountering a branch instruction.

Library functions and inline assemblies. Although the kernel does not have the notion of standard libraries like `libc`, common functionalities such as memory allocation are abstracted out, and most of them reside in the `lib` directory in the kernel source tree. These library functions can be generally categorized into five types:

- memory allocations (e.g., `kmalloc`),
- memory operations (e.g., `memcpy`),
- string operations (e.g., `strnlen`),
- synchronization operations (e.g., `mutex_lock`), and
- debug and error reporting functions (e.g., `printk`).

We choose to not let DEADLINE symbolically execute into these functions; instead, we manually write symbolic rules for each of these functions to capture the interactions between their function arguments and return values symbolically. Fortunately, for the purpose of *double-fetch bug* detection, there are only 45 and 12 library functions we need to handle for the Linux and the FreeBSD kernel, respectively, which incurs a reasonable amount of manual effort.

In terms of inline assemblies, although they are commonly found in kernel code, not many of them are related to *double-fetch bug* detection and hence will be filtered out early without showing in the execution path. For those that commonly appear in the execution paths (e.g., `bswap`), we write manual rules to approximate their effects on the symbolic values and ignore the rest, i.e., assuming them to have no effects.

4.4.3 Memory model

Traditional symbolic executors model memory as a linear array of bits or bytes and rely on the *select-store axioms* and its extensions [103, 104] to represent memory read and write. The select expression, `select(a, i)`, returns the value stored at position `i` of the array `a` and hence models a memory read, while a `store(a, i, v)` returns a new array identical to `a`, but on position `i` it contains the value `v` and hence models a memory write. This model has been proven successful by symbolic executors like KLEE [48] and SAGE [49]. However, it cannot be directly applied in DEADLINE for *double-fetch bug* detection.

One missing piece in this memory model is that two reads from the same address are assumed to always return the same value if there is no `store` operation to that address between the reads. While this is true for single-threaded programs (or multi-threaded programs with interleavings flattened), it does not hold for userspace accesses from kernel code, as a user process might change the value between the two reads, but those operations will not be backed by a `store` in the trace. In fact, if DEADLINE adopts this assumption, it would never find a *double-fetch bug*.

To address this issue, DEADLINE extends the model by encoding a monotonically increasing epoch number in the reads from userspace memory to represent that the values copied in at different fetches can be different. However, for kernel memory reads, DEADLINE does not add the epoch number and does assume that every load and store to the address is enclosed in the execution path. Otherwise, it becomes a kernel race condition, which is out of the scope for DEADLINE and is assumed to be nonexistent. To infer whether a pointer points to userspace or kernel memory, DEADLINE relies on the `__user` mark and considers that any pointer marked as `__user` is a userspace pointer (e.g., variable `uattr` in line 2, Figure 4.1a), and a pointer without the `__user` mark points to kernel memory (e.g., variable `attr`, in line 3, Figure 4.1a).

Another extension DEADLINE has to make to the memory model is that instead of assuming the whole memory to be an array of bytes (or bits), DEADLINE uses an array of

```
static int not_buggy1
  (int __user *uptr1,
   int __user *uptr2) {
  // uptr1 <-- UMEM(0)
  // cannot prove
  // uptr2 == uptr1, so
  // uptr2 <-- UMEM(1)

  int x1, x2;
  get_user(x1, uptr1);
  if(x1 == 0)
    return -EINVAL;

  get_user(x2, uptr2);
  return x2;
}
```

```
static void *not_buggy2
  (struct request __user *up,
   struct request *kp) {
  // up <-- UMEM(0)

  void __user *ubuf, void *kbuf;
  copy_from_user(kp, up, sizeof(*kp));
  if(!kp->buf)
    return -EINVAL;

  ubuf = kp->buf;
  // cannot prove ubuf == up, so
  // ubuf <-- UMEM(1)
  kbuf = memdup_user(kp->buf, kp->len);
  return kbuf;
}
```

Figure 4.3: DEADLINE's memory model cannot prove that the two fetches come from the same userspace object. Therefore, these two cases will not be considered as *double-fetch bugs*.

bytes to represent each single memory object and maps each pointer to one memory object. DEADLINE uses a few empirical rules to create this mapping:

- Different function arguments or global variables are assumed to be pointing to different memory objects (if they are pointers or integers that can be casted to pointers);
- Newly allocated pointers (via `kmalloc`, etc) are assumed to be pointing to new memory objects;
- When an assignment occurs, the object is also transferred (e.g., assigning a function argument to a local variable), meaning that the local variable is pointing to the same object as the argument. In fact, this is implicitly handled by the SSA form of the LLVM IR;
- For any other pointer, if we cannot prove that its value falls in the range of any existing object, assume it points to a new object. For example, a pointer like (`&req->buf`) is considered as pointing to a sub-range of the `req` object, while `req->buf` is considered as pointing to a new object.

Furthermore, when checking for *double-fetch bugs*, DEADLINE considers only the cases where the address pointers are pointing to the same userspace object. For example, in both cases shown in Figure 4.3, DEADLINE cannot prove that the two fetches come from the same userspace memory object. Therefore, DEADLINE does not mark them as *double-fetch* *bugs*. This design decision is made based on some implicit programming practices. For example, there is no need to pass in two pointers that point to the same memory region (i.e., the case of `uptr1 == uptr2` on the left side of Figure 4.3); or it is very uncommon to copy from cyclic buffers (i.e., the case of `up->buf == up` on the right side of Figure 4.3). However, in the case where DEADLINE can prove that two pointers have the same value, the memory object reference is also transferred, as shown in the case of Figure 4.2c, line 20.

4.4.4 Checking against the definitions

Upon finishing the translation from IR to SR, DEADLINE invokes the SMT solver to check whether all conditions listed in section 4.1 can be met.

DEADLINE first checks whether the two fetches, $F_0 =< A_0, S_0 >$, $F_1 =< A_1, S_1 >$, share an overlapped memory region. To do this, on top of the path constraints (which are already added to the solver during symbolization), DEADLINE further adds the constraint $(A_1 \leq A_0 < A_1 + S_1 \quad || \quad A_0 \leq A_1 < A_0 + S_0)$ to the solver (line 18 in the running example Figure 4.1c). DEADLINE then asks the solver to check whether there exist any overlapped regions with all the assertions. An overlap is represented by a triple $< N, i, j >$ and should be interpreted as that byte i to j in userspace object N being copied into the kernel twice in this *multi-read*. In the running example, there is one overlap, $< 0, 4, 7 >$, as shown in line 20.

If there are no overlapped regions, this *multi-read* is considered safe. Otherwise, for each overlap identified, DEADLINE further checks whether there is control dependence or data dependence established based on this region:

- In the case of control dependence only, collect the constraints for $@N(i, j, U0)$ (denoted as C_0) and $@N(i, j, U1)$ (denoted as C_1) and prove that C_1 is the same as C_0 or is even more restrictive than C_0.
- In the case of data dependence, prove that $@N(i, j, U0) == @N(i, j, U1)$, as shown in line 23 of the running example.

- In the very rare cases where there is no relation found, there is a redundant fetch.

Depending on the result, DEADLINE marks the *multi-read* as safe if the above proofs succeed and a bug otherwise.

4.5 Implementation

DEADLINE is implemented as an LLVM pass (6,395 LoC) based on LLVM version 4.0 and uses Z3 [73] version 4.5 as its theorem prover. The rest of this section covers the most important engineering problems we solved when developing DEADLINE, including maximizing code coverage, compiling and linking kernel source into LLVM IR, as well as program slicing and loop unrolling algorithms used in execution path construction.

Maximize code coverage. To detect *double-fetch bugs* for the whole kernel, we need to compile not only the kernel base but also as many modules as possible, including drivers, file systems, and peripheral modules that are rarely compiled in the generic configuration. In addition, within a source file, the actual code compiled is usually guided by many `#ifdef` statements. For example, the functions designed to bridge 32-bit applications with 64-bit kernels will be compiled only when `CONFIG_COMPAT` is enabled. We would like to cover these functions too.

To do this, we modify the configuration process for both the Linux and the FreeBSD kernels. For Linux, we rely on the built-in `allyesconfig` setting, which effectively enables all `CONFIG_*` macro (more than 10,000 items). Similarly, for FreeBSD, we rely on the `make LINT` command to output all available options and enable them all to get the build configuration file.

Compiling source code to LLVM IR. Since the Linux kernel is not yet compatible with the LLVM tool-chain, we compile it with the following steps: 1) we first build the kernel with GCC and collect the build log; 2) we then parse the log to extract compilation flags (e.g., `-I`, `-D`) for each source file and feed the flags to Clang to compile the file again to LLVM IR; 3) we again use the linking information in the build log and use `llvm-link` to merge the generated bitcode files into a single module. Files that are incompatible with LLVM will fail in step 2, which are only eight (out of 15,912 files in Linux 4.13.2).

For the FreeBSD kernel, although it can be successfully compiled with Clang, we cannot directly add the *-emit-llvm* flag to generate LLVM IR because the compilation process checks whether the generated object files are ELF files and will abort if not. Therefore, similar to the Linux kernel compilation, we compile the FreeBSD kernel in the normal way, parse the build log, re-compile the files to IR, and merge them into a single module.

Slicing and stitching. In order to obtain execution paths to feed to the symbolic engine, we do *backward slicing* to get sensitive instructions that may affect or constraint the address and size argument of transfer functions and do *forward slicing* to get sensitive instructions that may be affected by the fetch-in values of transfer functions, respectively. We stitch the sensitive instructions to construct paths that are possibly executed at runtime. algorithm 3 shows our slicing and stitching algorithm: the input is a function F that contains a pair of call-sites, $C1$ and $C2$, which invoke transfer functions; the output is a set of paths (i.e., P) that contain only sensitive instructions. Pseudo-code (i.e., line 4 – 16) shows the backward slicing algorithm: starting from the parameters of the call-sites, which are initialized as the seed vector V_v, we recursively identify all sensitive instructions (i.e., S_i) that may affect the values in V_v according to data and control dependencies, and recursively update V_v according to the use-def chains. We also use a set S_v to record visited values to avoid revisiting the same values. Similarly with the backward slicing, pseudo-code (i.e., line 17 – 30) shows the forward slicing algorithm, in which the instructions affected by the arguments are obtained by checking a value v's $users$, which can be a `LoadInst` loading data from v, or a `BranchInst` using v in its condition, etc. With all sensitive instructions (i.e., S_i) generated from forward and backward slicing, we refine F's CFG by cutting off the instructions not in S_i (i.e., line 31), which provides a refined CFG used for constructing paths (i.e., line 32). We follow original control flows to construct paths that are possibly executed at runtime.

Algorithm 3: do_slicing_and_stitching(F, $< C1, C2 >$)

```
In  : $F$ - A function contains a double-fetch pair ¡C1, C2¿
In  : $< C1, C2 >$ - A double-fetch pair callsites in $F$
Out : $P$ - A set of paths
1  $S_i \leftarrow \emptyset$
2  $S_v \leftarrow \emptyset$
3  $G \leftarrow F$'s CFG
   /* do backward slicing to identify instructions that
      can affect the double-fetch pair                    */
4  $V_v \leftarrow C1$.params $\cup$ $C2$.params
5  while !$V_v$.empty() do
6      $v \leftarrow V_v$.pop()
7      $S_v$.insert($v$)
8      if $v$.isInst() then
9          $S_i$.insert($v$)
10         for Use $u$ : $v$.operands() do
11             if !$S_v$.find($u$) then
12                 $V_v$.append($u$)
13             end
14         end
15     end
16 end
   /* do forward slicing to identify instructions that are
      affected by the double-fetch pair                   */
17 $V_v \leftarrow C1$.params $\cup$ $C2$.params
18 $S_v$.clear()
19 while !$V_v$.empty() do
20     $v \leftarrow V_v$.pop()
21     $S_v$.insert($v$)
22     if $v$.isInst() then
23         $S_i$.insert($v$)
24     end
25     for User $u$ : $v$.users() do
26         if !$S_v$.find($u$) then
27             $V_v$.append($u$)
28         end
29     end
30 end
31 refineCFG($G$, $S_i$)
32 $P \leftarrow$ getPaths($G$)
```

Loop unrolling. It is impossible to statically obtain all paths when a program contains unbounded loops. Even for the loops with fixed bounds, exploring all paths is inefficient. Therefore, we unroll each loop n times (n is configurable) to cover as many runtime paths as possible. algorithm 4 shows how we unroll loops in a CFG. In procedure $merge$: for each loop in a CFG, if the loop does not contain embedded loops, we merge all instructions inside the loop as a single node, which simplifies the CFG to a directed acyclic graph (DAG), in which we can directly get all paths. Procedure $unroll$ takes in a path from the DAG, and recursively unroll the loop nodes for n times until there are no loop nodes on the path. Although this unrolling algorithm is simple, it is proved to be efficient and effective.

Algorithm 4: merge_and_unroll

```
Function merge ($cfg$):
    while $cfg.hasLoop()$ do
        for Loop loop : $cfg$.loops() do
            if loop.isAtomic() then
                $loop$.merge()
            end
        end
    end
Function unroll ($path$, $n$):
    $changed \leftarrow true$
    while $changed$ do
        $changed \leftarrow false$
        for Node node : $path$ do
            if $node$.isLoopNode() then
                for int $i = 0; i < n; i = i + 1$ do
                    $node$.unroll()
                end
                $changed \leftarrow true$
            end
        end
    end
```

Table 4.1: Distribution of *multi-reads* and *double-fetch bugs* found by DEADLINE in the Linux and FreeBSD kernels

Component	**# Multi-Reads**		**# Double-fetch Bugs**	
	Linux	**FreeBSD**	**Linux**	**FreeBSD**
Core modules	25	4	2	0
Drivers	760	86	16	0
Filesystem	246	9	2	1
Networking	73	2	3	0
Total	1,104	101	23	1

4.6 Findings

In this section, we show DEADLINE's performance in both detecting *multi-reads* and *double-fetch bugs* in kernel software. Table 4.1 summarizes the number of *multi-reads* detected in the Linux and FreeBSD kernels and how many of them are actually *double-fetch bugs*.

4.6.1 Detecting *multi-reads*

This experiment is conducted on version 4.13.3 for the Linux kernel and 11.1 (July, 2017 release) for the FreeBSD kernel. As shown in Table 4.1, DEADLINE reports 1,104 *multi-reads* in the Linux kernel and 101 *multi-reads* in the FreeBSD kernel, as FreeBSD has a much smaller codebase. Furthermore, besides device drivers which have been studied in prior works [43, 44], many other kernel components, including the core modules (e.g., `ipc`, `sched`, etc), might issue multiple fetches from userspace, and some of them can be buggy.

More importantly, the scale of 1,104 *multi-reads* is not suitable for manual verification, not to mention keeping up with the frequent kernel updates. Therefore, this finding supports the claims that formal definitions are needed to define when a *multi-read* turns into a *double-fetch bug* and that automatic vetting is needed to alleviate this manual effort. This motivates the development of DEADLINE.

4.6.2 Detecting and reporting *double-fetch bugs*

Confirming previously reported bugs. We first show that DEADLINE is at least as good as prior works in detecting *double-fetch bugs*. In particular, DEADLINE runs against Linux kernel 4.5, the same version Wang *et al.* [44] used in their work. Out of five bugs reported in [44], DEADLINE found four of them, including `vop_ioctl`, `audit_log_single_ execve_arg`, `ec_device_ioctl_xcmd`, and `ioctl_send_fib`. DEADLINE is unable to detect `sclp_ctl_ioctl_sc` as DEADLINE compiles the kernel for the x86 architecture while `sclp_ctl.c` is only compilable for the IBM S/390 architecture.

Finding new bugs. A more important task for DEADLINE is to find new bugs. This experiment is conducted on version 4.12.7 to 4.13.3 for the Linux kernel and 11.1 (July, 2017 release) for the FreeBSD kernel [2].

Out of all *multi-reads* found in the kernels, DEADLINE detected 23 *double-fetch bugs* in Linux and one bug in FreeBSD. We manually checked all the bugs and reported them to the kernel maintainers. The full list of detected *double-fetch bugs* are shown in Table 4.2. At the time of writing:

- Nine bugs have been fixed with the patches we provided.
- Four bugs are acknowledged. We are currently working with the kernel maintainers to finalize the patches.
- Nine bugs are pending for review but no confirmation has been received.
- Two bugs are considered as "won't fix," as the maintainers do not think they are exploitable right now.

In summary, the number of reported bugs is significantly higher than in prior works (six in Linux and zero in FreeBSD). More importantly, while DEADLINE found significantly

[2]We perform bug finding iteratively as we develop and improve DEADLINE, which explains why several versions of Linux kernel are used.

Table 4.2: A listing of *double-fetch bugs* found and reported. In the *complication* column, we anticipate the reasons why the bug cannot be found by prior works. For 18 bugs that we submit patches for, we also list the strategy we use to fix the bugs, which is discussed in detail in subsection 4.6.4. For the remaining six bugs, the patching is likely to require a lot of code refactoring and we are working with the kernel maintainers to finalize a solution.

#	File	Function	Status	Complication	Patching Strategy
1	block/scsi_ioctl.c	sg_scsi_ioctl	Acknowledged	Macro expansion	Incremental copy
2	drivers/acpi/custom_method.c	cm_write	Submitted	-	Value override
3	drivers/hid/uhid.c	uhid_event_from_user	Won't Fix	Macro expansion	Abort on change
4	drivers/isdn/i4l/isdn_ppp.c	isdn_ppp_write	Patched	-	Single-fetch
5	drivers/message/fusion/mptctl.c	__mptctl_ioctl	Submitted	Inter-proc / Pattern	-
6	drivers/nvdimm/bus.c	__nd_ioctl (1)	Patched	Indirect call	Single-fetch
7	drivers/nvdimm/bus.c	__nd_ioctl (2)	Acknowledged	Indirect call	-
8	drivers/scsi/aacraid/commctrl.c	aac_send_raw_srb	Submitted	-	Value override
9	drivers/scsi/dpt_i2o.c	adpt_i2o_passthru	Submitted	-	-
10	drivers/scsi/megaraid/megaraid.c	mega_m_to_n	Submitted	Unknown pattern	Single-fetch
11	drivers/scsi/megaraid/megaraid_mm.c	mraid_mm_ioctl	Submitted	Inter-procedural	-
12	drivers/scsi/mpt3sas/mpt3sas_ctl.c	_ctl_getiocinfo	Patched	Inter-procedural	Single-fetch
13	drivers/staging/lustre/lustre/llite/llite_lib.c	ll_copy_user_md	Won't Fix	-	Override
14	drivers/tty/vt/vt.c	con_font_set	Patched	Loop / Pattern	Single-fetch
15	drivers/vhost/vhost.c	vhost_vring_ioctl	Submitted	Inter-procedural	-
16	fs/coda/psdev.c	coda_psdev_write	Acknowledged	Unknown pattern	Value override
17	fs/nfsd/nfs4recover.c	cld_pipe_downcall	Acknowledged	-	Value override
18	kernel/events/core.c	perf_copy_attr	Patched	-	Value override
19	kernel/sched/core.c	sched_copy_attr	Submitted	-	Value override
20	net/compat.c	cmsghdr_from_user_compat_to_kern	Patched	Loop involvement	Abort on change
21	net/tls/tls_main.c	do_tls_setsockopt_tx	Patched	Unknown pattern	Single-fetch
22	net/wireless/wext-core.c	ioctl_standard_iw_point	Submitted	-	-
23	sound/pci/asihpi/hpioctl.c	asihpi_hpi_ioctl	Patched	-	Value override
24	netsmb/smb_subr.c	smb_strdupin	Patched	Unknown pattern	Single-fetch

more *multi-reads*, it further automatically looks for real *double-fetch bugs* in the haystack of *multi-reads*, which is otherwise beyond the scale of manual verification. Furthermore, we anticipate that 14 out of the 24 bugs DEADLINE found could never be found by prior works because of the complications in the bugs, such as falling out of the empirical bug patterns, requiring inter-procedural analysis, loop involvement, and that a function is guarded by `#ifdef` macros.

Bugs marked as "won't fix". We pay special attention to the two bugs rejected for fixing by the developers, as they represent potential false alarms by DEADLINE as well as show the limitations of DEADLINE.

In the case of `uhid_event_from_user`, the developers actually acknowledged that the race condition can occur in userspace; however, they do not believe that this can cause serious harm, as quoted by one of the maintainers: *"With current code, worst case scenario is someone short-cutting the compat-conversion by setting UHID_CREATE after uhid_-event_from_user() copied it. However, this does no harm. If user-space wants to shortcut the conversion, let them do so..."*

In the case of `ll_copy_user_md`, DEADLINE falsely reports it due to an assumption on an enclosing function. By constructing execution paths within the enclosing function only, DEADLINE implicitly assumes that if there is an *overlapped-fetch*, careful developers should finish checking that the doubly-fetched values are either the same or subject to the same constraints. In this case, the checking should be `ll_lov_user_md_size(*kbuf) == lum_size` right after the second fetch. Otherwise, once the function returns, the developers lose the opportunity re-assert this relation. However, this implicit assumption does not hold in this case, as the derived value of the first fetch, `lum_size`, is passed out of the function as a return value, and the result of the second fetch, `kbuf`, is passed out by pointer. In other words, even outside this enclosing function, the relation between these two fetches can still be re-checked.

Bug distribution. Aligned with prior research [43, 44], a majority of *double-fetch bugs* are found in the driver code, indicating that drivers are still the most error-prone part in the kernel. This also aligns with the distribution of *multi-reads* where a majority of the *multi-reads* are located in drivers. However, file systems, networking components, or even the core kernel might also be subject to *double-fetch bugs*.

Detection time. On a machine with Intel Xeon E5-1620 CPU (four cores) and 64GB RAM running 64-bit Ubuntu 16.04.3 LTS, DEADLINE finishes detection in four hours for the Linux kernel and one hour for the FreeBSD kernel. Around 20% of the execution time is spent on finding *multi-reads* with static analysis and 80% of the time is spent on symbolic checking on these *multi-reads*.

4.6.3 Exploitation

```
char *smb_strdupin
  (char *s, size_t maxlen) {

  char *p, bt; int error;
  size_t len = 0;
  // test and check user strlen
  for (p = s; ;p++) {
    if (copyin(p, &bt, 1))
      return NULL;
    len++;
    if (maxlen && len > maxlen)
      return NULL;
    if (bt == 0)
      break;
  }
  p = malloc(len, M_SMBSTR);
  // copy the whole string
  error = copyin(s, p, len);
  if (error) {
    free(p, M_SMBSTR);
    return NULL;
  }
  // BUG: p is not NULL-termed
  return p;
}
```

(a) Buggy function

```
// syscall entry point
entry: ioctl() {
  ...
  // dispatch to device ioctl
  nsmb_dev_ioctl() {
    ...
    // dispatch by command
    smb_usr_t2request() {
      ...
      // [!] double-fetch bug
      buf = smb_strdupin();

      ...

      smb_t2_request() {
        ...
        smb_t2_request_int() {
          ...
          // [!] exploitation
          nmlen = strlen(buf);
        }
      }
    }
  }
}
```

(b) Call stack for an exploit

Figure 4.4: An exploitable *double-fetch bug* in the FreeBSD kernel. 4.4a shows the function flagged as buggy by DEADLINE and 4.4b shows the end-to-end call stack in the kernel if a user thread tries to exploit this bug by issuing an `ioctl` syscall.

Exploiting *double-fetch bugs* can be profitable but also challenging. Prior works [43, 44] have identified several ways to exploit a *double-fetch bug* in kernel.

Leaking information. This exploitation typically occurs in a process that does data transfer both to and from userspace, i.e., a request-response situation, as shown in the case of CVE-2016-6130. The bug in CVE-2016-6130 is very similar to the bug in `perf_copy_attr` (Figure 4.1), where the first fetch sanity checked the `size` value while the second fetch assumes `size` does not change and omitted the sanity check. Later, when the response is copied back to userspace based on the unchecked `size` value, a large chunk of kernel memory will be copied, hence causing a kernel information leak.

Bypassing restrictions. This exploitation typically occurs when the kernel wants to early reject a request from userspace. For example, in the `tls_setsockopt` case (Figure 2.3), a malicious user process can bypass the TLS version checking (line 9) by exploiting this double-fetch behavior although the intention of the kernel developers is to reject such requests.

Denial-of-service (DoS). This exploitation typically occurs when a memory operation, e.g., buffer allocation, memory compare, string operations, etc, is affected by the double-fetch procedure. For example, in the case of `smb_strdupin` (Figure 4.4), it is incorrect to assume that the string copied in after the second fetch is NULL-terminated and later applying the `strlen` on the string is likely to cause an overread into invalid kernel memory regions.

DEADLINE does not attempt to automatically reason about the exploitability of *double-fetch bugs* for two reasons: 1) Unlike memory errors that raise a definitive signal upon exploitation, e.g., an invalid memory access causing a crash, *double-fetch bug* exploitations do not usually raise such a signal and might have to rely on manually defined rules to measure whether the exploit succeeds. 2) Even if we could define all the exploitation rules, constructing them could still be a challenge, as the exploitation point is usually far from to the bug point. In the example shown in Figure 4.4, the exploitation point `strlen` is two function calls away from the buggy function. In this case, in order to construct an end-to-end exploit, DEADLINE needs to symbolically execute the whole `ioctl` syscall, which would take significantly longer if ever possible. More importantly, even if a *double-fetch bug* is

```
 kernel/events/core.c | 2 ++
 1 file changed, 2 insertions(+)

diff --git a/kernel/events/core.c b/kernel/events/core.c
index ee20d4c..c0d7946 100644
--- a/kernel/events/core.c
+++ b/kernel/events/core.c
@@ -9611,6 +9611,8 @@ static int perf_copy_attr(struct perf_event_attr __user *uattr,
    if (ret)
        return -EFAULT;

+   attr->size = size;
+
    if (attr->__reserved_1)
        return -EINVAL;
```

Figure 4.5: The patch to `perf_copy_attr` follows the override strategy

not exploitable right now, it does not mean that it will remain secure in the future. Careless code updates can easily turn a non-exploitable *double-fetch bug* into an exploitable one, as shown in the case of CVE-2016-5728.

4.6.4 Mitigation

Based on our experience in patch creation and our communications with kernel maintainers, there are in general four ways to patch a *double-fetch bug*.

Override with values from the first fetch. In this case, we simply ignore the value copied in during the second fetch and override it with the value from the first fetch. An example is shown in Figure 4.5, which is actually the patch to Figure 4.1. By doing so, we ensure that both the control dependence and data dependence established between these fetches are preserved.

Abort if changes are detected. In this case, we add a sanity check after the second fetch to ensure that the intended relation between the two fetches is honored by the user process, as shown in the example of Figure 4.6, which is actually the patch to Figure 4.2.

Incremental fetch. In this case, we intentionally skip the bytes copied in during the first fetch. In other words, we start the second fetch from an offset equal to the length of the first fetch. An example is shown in Figure 4.7. By doing this, these two fetches are now from non-overlapped userspace memory regions and will never constitute a *double-fetch bug*.

```
 net/compat.c | 7 +++++++
 1 file changed, 7 insertions(+)

diff --git a/net/compat.c b/net/compat.c
index 6ded6c8..2238171 100644
--- a/net/compat.c
+++ b/net/compat.c
@@ -185,6 +185,13 @@ int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg, struct sock *sk,
         ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
     }

+    /*
+     * check the length of messages copied in is the same as the
+     * what we get from the first loop
+     */
+    if ((char *)kcmsg - (char *)kcmsg_base != kcmlen)
+        goto Einval;
+
     /* Ok, looks like we made it.  Hook it up and return success. */
     kmsg->msg_control = kcmsg_base;
     kmsg->msg_controllen = kcmlen;
```

Figure 4.6: The patch to `cmsghdr_from_user_compat_to_kern` follows the abort on change strategy

Refactor into a single-fetch. If there is only control dependence between the two fetches, we could reduce this double-fetch behavior into a single-fetch. This approach generally improves the performance as we eliminate one fetch but might result in more lines of code, as now we need to multiplex the `if` checks every time a fetch occurs.

In principle, all these strategies have the same effect—preventing a *double-fetch bug* from being exploited. However, which strategy is taken for a specific *double-fetch bug* is usually based on case-by-case considerations such as performance concerns, number of lines changed, accordance with existing sanity checks, and the maintainers' personal preferences. Besides, several bugs cannot be patched within 50 lines of change due to the complications in current codebases. We are working with the maintainers to finalize the patch.

Preventing exploits with transactional memory. As the root cause of a *double-fetch bug* is the lack of atomicity and consistency in userspace memory accesses across fetches, transactional memory (e.g., Intel TSX) can be a generic solution. Conceptually, one could mark transaction start before the first fetch and mark transaction end after the second fetch. If a race condition occurs, the transaction will abort and the kernel will be notified.

```
 block/scsi_ioctl.c | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

diff --git a/block/scsi_ioctl.c b/block/scsi_ioctl.c
index 7440de4..8fe1e05 100644
--- a/block/scsi_ioctl.c
+++ b/block/scsi_ioctl.c
@@ -463,7 +463,13 @@ int sg_scsi_ioctl(struct request_queue *q, struct gendisk *disk, fmode_t mode,
     */
    err = -EFAULT;
    req->cmd_len = cmdlen;
-   if (copy_from_user(req->cmd, sic->data, cmdlen))
+
+   /*
+    * avoid copying the opcode twice
+    */
+   memcpy(req->cmd, &opcode, sizeof(opcode));
+   if (copy_from_user(req->cmd + sizeof(opcode),
+               sic->data + sizeof(opcode), cmdlen - sizeof(opcode)))
        goto error;

    if (in_len && copy_from_user(buffer, sic->data + cmdlen, in_len))
```

Figure 4.7: The patch to `sg_scsi_ioctl` follows the incremental copy strategy

DECAF [105] is a proof-of-concept based on these insights. However, it over-simplifies the kernel code by failing to consider the cases of 1) false aborts due to large memory access footprint (which is very likely for *multi-reads*), 2) multiple exit points in the syscall execution (e.g., returning before second fetch), and 3) mixing of TSX-enabled and non-TSX code (e.g., a function can be called within or without a transactional context). Furthermore, DECAF still requires the developers to manually inspect and instrument kernel code. Therefore, to make the TSX-based solution practical, these technical challenges should be addressed and *automated* integration of TSX APIs and kernel code is necessary.

4.7 Discussion and Limitations

Checking beyond kernels. It is worth-noting that *double-fetch bugs* are not specific to kernels. In theory, it might exist in software systems in which 1) the memory region is separated into sub-regions with various levels of privilege and 2) multi-threading is supported. Therefore, software systems beyond kernels such as Xen, SGX, and even userspace programs like the Chrome browser are also subject to *double-fetch bugs*.

To apply DEADLINE to these software systems, we need to clearly identify the boundary of privileges and the interfaces for transferring data from a low-privilege memory region to a high-privilege memory region. That is, DEADLINE requires pre-defined "fetching" interfaces. Fortunately, we observed that privileged software systems typically have limited interfaces for fetching data from low-privilege to high-privilege memory regions. This is arguably to better maintain the boundary of separated regions. As such, we believe that it is feasible to collect "fetching" interfaces with reasonable engineering effort.

We further discuss the limitations of DEADLINE from three aspects:

- which part of kernel source code DEADLINE cannot cover,
- what kind of execution paths DEADLINE cannot construct for *multi-reads*, and
- when the symbolic checking for *double-fetch bugs* fails.

Note that many of these limitations will be addressed in section 4.8.

Source code coverage. Although DEADLINE covers a majority of kernel codebase, there are two cases DEADLINE currently cannot handle:

1) Files not compilable under LLVM cannot be analyzed by DEADLINE. For Linux 4.13.2, they include three file system files and four driver files, which are likely to contain both *multi-reads* and *double-fetch bugs*. We believe this will not be addressed soon with the synergy between the kernel and LLVM community.

2) Although DEADLINE enables all the config options during compilation, DEADLINE certainly misses the code pieces that are compiled when a `CONFIG_*` should be disabled. However, a complete solution would require tweaking `Y` and `N` for over 10000 config items, which is unrealistic.

Path construction. DEADLINE aims to find all execution paths associated with a *multi-read*. However, due to the complexity in kernel code, DEADLINE's path construction has three limitations:

1) DEADLINE enforces a limit (currently 4096) to the number of execution paths constructed within an enclosing function. Although in most of the cases there are less than 100

paths, we did observe 17 functions that exceed this limit. Therefore, DEADLINE could have missed *double-fetch bugs* should they exists in those unconstructed paths.

2) DEADLINE also enforces a limit (currently 1) to the loop unrolling, with the assumption that fetches in loops are usually designed in an incremental manner. However, this assumption might be wrong and the fetch inside the loop itself makes a *double-fetch bug* when the loop is unrolled multiple times. Furthermore, there could be cases when cross-loop *double-fetch bugs* occur when a loop is unrolled to a specific time. Although we believe both cases are rare, we cannot prove that they do not exist in kernel.

3) If there is a branch inside a loop, DEADLINE picks only one subpath to unroll the loop. However, there might be cases when a *double-fetch bug* occurs when the sub-paths are taken in a specific order when unrolling the loop multiple times, e.g., the true branch is taken in the first unrolling and the false branch is taken in the second unrolling.

Symbolic checking. DEADLINE symbolic checker is limited by how well we model complicated code pieces like assemblies and library function calls as well as the assumptions in the memory model.

1) DEADLINE ignores a majority of inline assemblies, i.e., assuming they have no impact on the symbolic checking. This could lead to missing constraints or incomplete SR assignment, especially when the assemblies issue memory operations. In addition, for the library functions that we manually write rules for, there might be imprecision. For example, we assume that `strnlen` might return any value between 0 and the `len` argument, but actually it is also constrained by the string buffer, which we do not model in the rule.

2) DEADLINE's empirical mapping from pointers to memory object might not reflect the actual situation. As shown in Figure 4.3, if the function calling `not_buggy1` already ensures that `utrp1 == uptr2`, or the `struct request` is designed in such a way that `up->buf == up`, then it would be wrong for DEADLINE to treat them as non-buggy.

3) DEADLINE's assumption about the enclosing function might be incomplete. As shown in the "won't fix" case `ll_copy_user_md`, it is possible that the information to

assert the relations established between the two fetches are passed out of the enclosing function and re-checked elsewhere.

4.8 Future work: Towards Comprehensive Checking with C-to-SMT Transpilation

Although DEADLINE sheds lights on adapting conventional symbolic execution to concurrent programs, it is far from a sound and complete symbolic execution framework as many compromisations are made to cope with the size and complexity of kernel code. Following are the major blockades that prevent DEADLINE from being sound and complete:

- The CFG exploration strategy suffers from path explosion: in theory, due to the depth-first search nature of DEADLINE's path construction algorithm, the total number of paths to be symbolic executed has a complexity of $2^{\#branches}$. DEADLINE only samples a few paths in face of path explosion.

- Loops are unrolled to a certain depth only: essentially, for 1) loops with a large iteration count, or even worse, 2) unbounded loops with statically unknown iteration count, the checking in DEADLINE is incomplete, which might lead to missing bugs (*e.g.*, bugs only show up after certain loop iterations).

- DEADLINE employs a simple but imperfect memory model, which is not suitable for modeling memory access patterns other than plain reads and writes. For example, several patterns we hope to address include: 1) memory allocations with symbolic sizes, 2) `memset`/`memcpy` with a symbolic length, and 3) Memory content partial override with path conditions. Although not a severe concern for *double-fetch bugs* detection, handling these patterns is essential for detecting many other concurrency bugs, including but not limited to, malicious data races.

- The concurrent memory versioning for DEADLINE applies to userspace memory accesses only while still assuming that memory accesses in kernel space is sequential. This limits DEADLINE to find only concurrency bugs that lives across the kernel-user boundary. To extend DEADLINE for finding concurrency bugs arised from memory accesses within the same address space, we need to apply memory versioning to virtually all memory accesses.

To systematically address these limitations, we envision KSA, a symbolic execution framework re-imagined for OS kernels. The mission of KSA is to derive a precise and lossless symbolic representation for (in theory) *each and every value* in a complete kernel component (*e.g.*, the whole file system module `ext4`). By doing so, follow-up operations, being them finding bugs or verifying properties, can be treated by composing and solving constraints based on the symbolic representations produced by KSA. Any value that cannot be modeled by KSA for practicality reasons belongs to an explicit list of incompleteness: *i.e.*, C programming constructs that we currently do not have a good theory to encode in KSA. In this way, KSA is at least sound and complete in the subset of C programming constructs that it can model.

Besides the more comprehensive modeling effort, another important distinction between KSA and DEADLINE is that: unlike DEADLINE, the focus of KSA is not end-to-end solvability, but devising a sound and complete representation of concurrent C programs which is suitable for SMT solvers to reason about. In other words, KSA aims for a lossless transpilation from C to SMT formulae. This separation of concerns allows KSA to focus on faithfully capturing the semantic of concurrent C programs without losing genericity (*e.g.*, representing arithmetic operations in two versions, one with integer theories and the other with bit-vector theories); while leaving formulae simplification, optimization, and constraint solving to the fast evolving SMT solvers.

Furthermore, in KSA, the tasks of information picking, filtering, and constraint composing are delegated to developers of specific bug checkers. The bug checker may also apply abstraction techniques to ease the solving of complex constraints. For example, forcing a limited depth of loop unrolling instead of using recursive functions to model unbounded loops. Relaxations on the constraints might also derive from the characteristics of the bug to be checked as well as the tolerance to false positives and false negatives. For example, an `unknown` answer from the SMT solver may be treated as *not-a-bug* (being conservative for bug reporting) or *is-a-bug* (being aggressive).

4.8.1 Assumptions and Rationale

As described in section 4.8, KSA aims to address the four issues that prevent DEADLINE from being sound and complete. The solution proposed in KSA is to derive a symbolic representation for (in theory) every value in a whole kernel component (*e.g.*, the file system module `ext4`). With such a symbolic representation, follow-up operations, being them finding bugs or verifying properties, can be treated by creating and solving constraints on these symbolic representations.

However, not any arbitrary C program can be transformed into symbolic formulae in KSA. KSA makes a few assumptions on the programs it might deal with:

1. All functions are materialized. This assumption allows KSA to execute into any function during symbolization process. As a result, creating function summaries is not necessary for KSA as whenever a function call is encountered, KSA is allowed to insert the CFG of the called function into the current CFG with existing execution context (*i.e.*, symbolic values). The only exception to the rule is library functions. Essentially, a limited number of library functions (*e.g.*, `malloc`-related memory management routines, `memset` and `memcpy` operations, etc.) can be declarations only and in these cases, KSA relies on built-in summaries to emulate their effects.

2. All global variables are defined. In other words, all global variables have proper initialization which is available statically. This assumption allows KSA to initialize and evolve all global variables properly. Again, as an exception to this rule, assemblies and linker-introduced constants (*e.g.*, section alignments, etc), can be omitted. In these cases, KSA might rely on built-in summaries to emulate their initial values.

3. Computationally intractable code can be skipped. A good example of computationally intractable code is cryptography functions or hashing, which can never be symbolically executed verbatim. In these cases, KSA relies on manual annotations to skip the symbolization of these functions.

Rationale. Essentially, these assumptions imply that there is no "external environment" for the kernel module to be analyzed by KSA, except a very limited number of library functions and constants. These assumptions might not hold in complex userspace programs (*e.g.*, browsers), as for them, the execution environment (*e.g.*, libraries linked into the program) might be too complicated to be decoupled. For example, in browsers, it is hard to decouple the JavaScript engine from the sandboxing mechanism or to decouple anything from `libc`. But OS kernels provide such an opportunity partially thanks to the strict coding standards enforced across various modules in the kernel.

4.8.2 Guarded Symbolic Representation

Being able to execute *sequentially* is the baseline of virtually any symbolic executor. Specifically, by using the term "sequential", we refer to the algorithm that 1) *exhaustively* explores the CFG of an arbitrary program (that fits our assumptions in subsection 4.8.1), 2) from entry to exit, and 3) in a single-threaded manner.

However, even for this basic requirement, the design space of a symbolic execution engine is vast. In this section, we first give our reasoning on the infamous path explosion problems commonly faced by any symbolic executor and try to comprehensively list the design choices made by KSA solve this problem. We would like to point out early that a common rationale behind all the design choices made by KSA is achieving both completeness and soundness. In other words, the goal of KSA is to keep *all possible information* about *all values* in the program. In particular,

- KSA does not assume a (set of) particular path(s) to enforce nor aims to maximize any particular coverage metric. We want KSA to be as generic as possible and provide all information in the subsequent checking phase.

- KSA does not invoke abstraction (e.g., abstract interpretation for branch merging or loop summarization) because this is a lossy way to represent program states.

Of course, being both sound and complete is a greedy goal and pressures the downstream for proper post-processing on the generated formulae. For example, individual bug checkers are now responsible for abstraction and formulae simplification; while extra burden is also added to constraint solvers for solving complicated constraints encoded with multiple theories. In essence, with separation of concerns, KSA is only responsible for the symbolization part while bug checkers and SMT solvers consume the symbolization.

The reason behind branching-induced path explosion

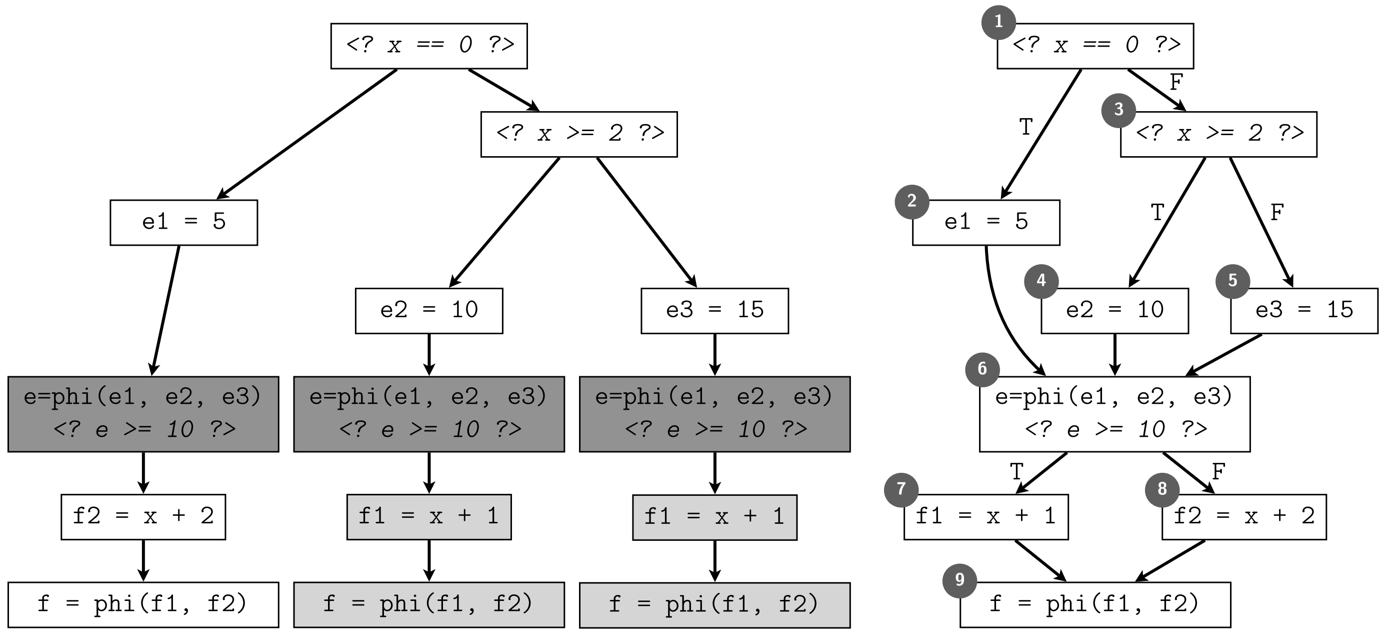
A common frustration on applying symbolic execution to large and complex software is the problem termed as *path explosion*. However, the reason for path explosion is usually three-fold, including 1) if-else branches, 2) unrolling unbounded loops, and 3) pointer aliases. All these problems will be addressed in KSA. This section focuses on the branching issue, in particular, to reason why branching poses scalability challenges for a DFS-based CFG exploration strategy and how to solve this problem with a BFS-based strategy.

Figure 4.8 shows a dummy function with three branches. Visualized in the CFG (see subfigure-c), an if-else branch creates two paths in the control flow. In order to explore both paths, a symbolic executor forks its internal states at the branching point (see subfigure-b). After forking, one child will inherit the true condition, *i.e.*, by asserting the condition to be true in its context, while the other child will inherit the false condition. Intuitively, the more branches a program have, the more forking is needed and the number of children grows exponentially as a result of this.

Therefore, although state forking is intuitive to reason and easy to implement, the scalability bottleneck has limited its usefulness in large programs. As illustrated, by forking at each branching point, the number of children to execute is linearly related to the number

```
int foo(int x) {
  if (x == 0) {
    e1 = 5;
  } else {
    if (x >= 2) {
      e2 = 10;
    } else {
      e3 = 15;
    }
  }
  e = phi(e1, e2, e3);
  if (e >= 10) {
    f1 = x + 1;
  } else {
    f2 = x + 2;
  }
  f = phi(f1, f2);
  return f;
}
```

(a) Program function



(b) DFS-based exploration strategy

(c) BFS-based exploration strategy

Figure 4.8: Illustration of the DFS- and BFS-based CFG exploration strategies. The DFS strategy (implied from the state forking approach) is used in major symbolic execution engines while the BFS strategy is used in KSA.

of possible paths in the CFG, which, in theory, is characterized by $2^{|V|}$ where $|V|$ stands for the total number of nodes (*i.e.*, basic blocks) in the CFG.

In the example in Figure 4.8, a total of three children are forked and basic block 6 and 8 are executed repeated in 3 and 2 forked children, respectively, as shaded in gray color in subfigure-b. More importantly, it is not hard to imagine that as the control-flow further grows, the amount of repetition will grow exponentially too, which will eventually cause the search tree shown in subfigure-b to explode.

A more philosophical reasoning about the path explosion is that forking at each if-else branch implicitly assumes a DFS strategy in exploring the CFG. In other words, by forking at branching points, at any point of time, the symbolic executor maintains only one context (*i.e.*, an append-only set of path conditions) and continue with that context as deep as possible. The symbolic executor only backtracks and changes context when the current context has been proved to be unsatisfiable.

A guarded representation for symbolic values

A nice property in the DFS-style CFG exploration is that every variable encountered will be definitive (subject to the current path constraints). In the running example (Figure 4.8), by forking at every branching point, whenever control reaches block 6 in each execution, the value for variable `e` is definitive, as it can only be 5, 10, and 15 in the three forked executions, respectively. However, this nice property comes at the cost of exploding the search space. A natural question is: can we re-balance the trade-off? More specifically, can we make the symbolic representation of variables more complicated but at the same time, break the exponential growth of complexity?

To answer the question, and inspired by prior works such as Veritesting [106], MultiSE [107], we introduce *guarded symbolic representation*. In a high level description, assuming that the program does not involve any memory operations (we will re-visit this in subsection 4.8.4),

- Each program variable has a corresponding symbolic representation named `SEValue`.
- Each `SEValue` maintains a set of instances, named `SEInstance`, where each represents one possible state that might be taken by a program value.
- Each `SEInstance` has an expression `expr` which represents the symbolic formula and a condition `cond`, which is a boolean predicate that represents the condition that have to be satisfied for the Value to take the symbolic `expr`.

As an illustration, the guarded symbolic representation of the running example (Figure 4.8) is shown in Table 4.3. Pay special attention to the representation of variables `e` and `f` in subtable-b. Both variables have multiple `SEInstances` and each instance is guarded by different conditions on the symbolic input `x`. It is worth noting that the reaching path is implicitly embedded in the guards. For example, there are two paths that could reach `f2 = x + 2`, one by `1-3-4-6-8`, baring condition $x \geq 2$, the other by `1-3-5-6-8`, baring condition $x < 2 \wedge x! = 0$. Merging and simplifying both condition get exactly the same guard $x \neq 0$ as KSA.

Such a guarded symbolic representation also opens the opportunities on checking based on the values of `e` and `f`. For example, one can compose a query asking whether we could ever reach to a state on `e + f == x + 6` which is only possible if `e == 5` $\wedge$ `f == x + 1`. However, there is no x that satisfies their guards on $x = 0 \wedge x! = 0$, therefore, we can safely conclude that the query `e + f == x + 6` is not reachable.

CFG path constraint tracking in a BFS way

Strictly speaking, the CFG exploration in KSA is more complicated than the basic BFS algorithm. In KSA, before trying to derive the reachability condition of a basic block, KSA needs to ensure that all its parents are visited first. This is an extra requirement not in the conventional BFS algorithm. However, this does assume that there is no loops in the CFG, (to be specific, no backedges), we will re-visit and relax this requirement in subsection 4.8.3.

The CFG exploration algorithm in KSA is presented in algorithm 5. Essentially, this

Table 4.3: Representing the running example (Figure 4.8) symbolically in KSA

(a) Path conditions on entering each basic block

BB	Reachability condition
1	$True$
2	$x = 0$
3	$x \neq 0$
4	$x \geq 2$
5	$x \neq 0 \wedge x < 2$
6	$x = 0 \vee x \geq 2 \vee (x \neq 0 \wedge x < 2) \implies True$
7	$x \neq 0$
8	$x = 0$
9	$x \neq 0 \vee x = 0 \implies True$

(b) Guarded symbolic representation of each variable

Variable	Option	Expr.	Guard
`e1=5`	0	5	$x = 0$
`e2=10`	0	10	$x \geq 2$
`e3=15`	0	15	$x \neq 0 \wedge x < 2$
`e=phi(e1, e2, e3)`	0	5	$x = 0$
`e=phi(e1, e2, e3)`	1	10	$x \geq 0$
`e=phi(e1, e2, e3)`	2	15	$x \neq 0 \wedge x < 2$
`f1=x+1`	0	$x + 1$	$x \neq 0$
`f2=x+2`	0	$x + 2$	$x = 0$
`f=phi(f1, f2)`	0	$x + 1$	$x = 0$
`f=phi(f1, f2)`	1	$x + 2$	$x \geq 0$

Algorithm 5: Reachability condition derivation for basic blocks

```
In  : $CFG$ - program control-flow represented in graph format
In  : $cond_{base}$ - the base condition for the $CFG$ entry block
Out : Reachability conditions for each $node$ in $CFG$
$root \leftarrow CFG$.entry()
$Q \leftarrow$ Queue($root$)
$H \leftarrow$ Set($root$)
$V \leftarrow$ Set()
while $Q$ is not empty do
    $node \leftarrow Q$.pop()
    if not all parents of $node$ is visited (i.e., in set $V$) then
        $Q$.queue($node$)
    else
        if $node$ has no parents (i.e., the entry block) then
            $node$.cond $\leftarrow cond_{base}$
        else
            $cond_{node} \leftarrow False$
            for each parent ($p$) of $node$ do
                if $p \rightarrow node$ is unconditional branch then
                    $rc_p \leftarrow True$
                    $cond_{node} \leftarrow cond_{node} \vee rc_p$
                else
                    for each instance ($i$) of $p$'s predicate do
                        if $p \rightarrow node$ is on True branch then
                            $rc_p \leftarrow i.guard \wedge i.expr$
                        else
                            $rc_p \leftarrow i.guard \wedge \neg i.expr$
                        end
                        $cond_{node} \leftarrow cond_{node} \vee rc_p$
                    end
                end
            end
            $node$.cond $\leftarrow cond_{node}$
        end
        $V$.add($node$)
        $Q$.queue(all children of $node$ that are not in $H$)
        $H$.add(all children of $node$ that are not in $H$)
    end
end
```

is a derivation procedure for the reachability condition for each basic blocks in the CFG. The basic idea is that, for any basic block, KSA will first derive the reachability condition for all its parent blocks first and OR-join their predicates that represent the reaching edges. A sample output from this algorithm is presented in Table 4.3, subfigure-a. Although the concept of augmenting symbolic expressions with guarding conditions is similar to MultiSE [107] and Veritesting [106], KSA does not alternate the execution mode of DFS and BFS. Instead, KSA enforces BFS strictly throughout the whole CFG. More importantly, KSA further brings the guarded representation mode into loop handling and memory modeling, which are not addressed in these two works.

4.8.3 Loop Modeling with Recursive Functions

The challenges caused by unbounded loops

In the literature on symbolic model checking, loops are generally treated as a roadblocker for making symbolic execution scalable for large and complex programs. However, it is worth-noting that not all loops pose challenges for symbolic executors. Figure 4.9 shows three types of loops and the bounded loop, *i.e.*, subfigure-a, can be well handled by existing symbolic executors. The reason is that for bounded loops, the number of iteration count is statically known. As a result, although each loop iteration runs into a branching point (*e.g.*, `i < 1000`), the predicate is always evaluated to `True` in the first 1000 iterations. Hence, not causing the symbolic executor to fork. Similarly, on the 1000th iteration, `i < 1000` will be `False` and the state does not need to be forked neither. In other words, the whole loop is implicitly linearized during the execution.

However, the same logic does not apply for the other two loops in Figure 4.9, as the termination condition of these loops depends on a statically unknown variable, `x`, which means that there is no statically computable iteration counts. In this case, per each loop iteration, conventional symbolic executors such as KLEE or SAGE are forced to fork, to explore the possibilities that, taking the example of subfigure-b, `x >= i` and `x == i + 1`.

```c
int loop_bounded(void) {
  int s = 0;
  for (int i = 0; i < 100; i++) {
    s += i;
  }
  return s;
}
```

(a) A bounded (fixed-iteration) loop

```c
int **loop_unbounded_2(int x) {
  int **matrix = malloc(x * sizeof(int*));
  for (int i = 0; i < x; i++) {
    matrix[i] = calloc(x, sizeof(int));
  }
  for (int i = 0; i < x; i++) {
    matrix[i][i] = x;
  }
  return matrix;
}
```

(b) An unbounded loop with memory operations

Figure 4.9: Illustration on bounded vs unbounded loops

This quickly leads to path explosion based on the how many times the loop may iterate. Even worse, the loop exploration might in theory, never stop as we can always keep unrolling the loop by increasing the value of `x`. As a result, existing works typically limit the unrolling to a certain depth, usually just a handful, to enforce a loop termination.

However, enforcing an arbitrary bound on the loops is not the ideal case and is far from the design goal of KSA, which is to have a faithful and lossless representation of every variable in the program. However, unbounded loops are extremely common in OS kernels. With a simple checking on the `btrfs` file system module in the Linux kernel, we found that around 80% of loops (4052 / 5124) are unbounded loops. This presses us to find a way to systematically symbolize every loop in a lossless manner.

Loop-to-recursion transformation and SMT-solving

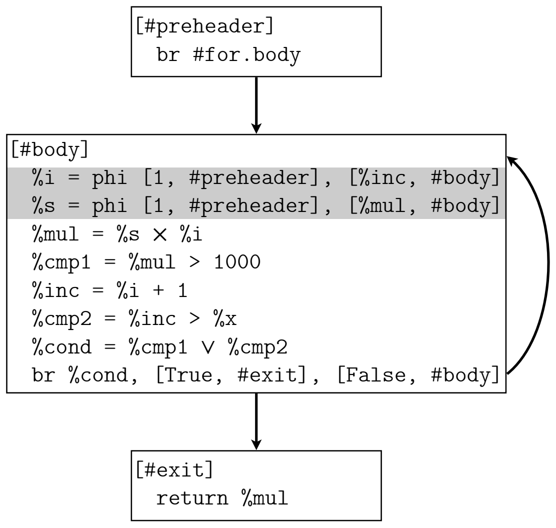
The key to symbolize loops losslessly is two-fold:

- Loops and recursive functions are equivalent constructs and can always be transformed programmatically.
- Modern SMT solvers are capable of solving recursive constraints, such as the muZ fixed-points engine shipped with Z3.

This section will shed light on how this can be done in KSA, with Figure 4.10 taken as a running example.

```c
int loop_unbounded_1(int x) {
  int s = 1;
  for (int i = 1; i <= x; i++) {
    s *= i;
    if (s > 1000) {
      break;
    }
  }
  return s;
}
```

(a) An unbounded loop



```
[#preheader]
  br #for.body

[#body]
  %i = phi [1, #preheader], [%inc, #body]
  %s = phi [1, #preheader], [%mul, #body]
  %mul = %s × %i
  %cmp1 = %mul > 1000
  %inc = %i + 1
  %cmp2 = %inc > %x
  %cond = %cmp1 ∨ %cmp2
  br %cond, [True, #exit], [False, #body]

[#exit]
  return %mul
```

(b) Control-flow graph of the loop part

Figure 4.10: An example of a simple unbounded loop and its CFG

Loop simplified form. The first thing to do is to transform a loop into a simplified form which eases the subsequent symbolization phase. In loop simplified form, each loop has

- A preheader that unconditionally branch into the loop body.
- A single backedge (which implies that there is a single latch block).
- Dedicated exits. That is, no exit block for the loop has a predecessor that is outside the loop. This implies that all exit blocks are dominated by the loop header.

The CFG shown in Figure 4.10 subfigure-b is a perfect illustration on loop simplified forms. Given the algorithm provided by the LLVM compiler [108], we verified that all loops in the Linux kernel can be transformed into this simplified form.

Transformation to recursive functions. A property derived from the loop simplification process is that all the loop *induction variables* are placed in the loop entry block, *i.e.*, the first block in loop body. In the running example, the loop body has only one basic block, and hence, that basic block is also the loop entry block and the induction variables are the two `phi` instructions highlighted in gray. By definition, a loop induction variable is a variable that has a base value when the loop is first entered and changes after every iteration. This means, that the induction variable has to be a `phi` instruction placed in the loop entry block.

In the running example, assuming we are on the k-th iteration of the loop, ($k = 0$ means when the loop is first entered), the recursion for the induction variables `i` and `s` can be expressed in the following formulas:

$$f_i(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_i(k-1) \ + \ 1 & \text{if } k > 0 \end{cases} \tag{4.1}$$

$$f_s(k) = \begin{cases} 1 & \text{if } k = 0 \\ f_s(k-1) \ \times \ f_i(k-1) & \text{if } k > 0 \end{cases} \tag{4.2}$$

Beyond the induction variables, an implicit but important fact we need to capture is that: if we manage to iterate the loop k times, then it must not exit in the prior $k-1$ iterations. This is essentially another recursive function expressed in the following formula:

$$f_{loop}(k) = \begin{cases} True & \text{if } k = 0 \\ f_{loop}(k-1) \ \wedge \ (f_i(k-1) \leq x) \ \wedge \ (f_s(k-1) \leq 1000) & \text{if } k > 0 \end{cases} \tag{4.3}$$

Modeling with a fixed-point engine. Although the abovementioned formulas are intuitive, they are not the best option to model for SMT solvers. The best way is to convert them into a single fixed-points recursive relation. Essentially, the relation is a function with signature $rel(indvars...) \rightarrow Bool$. In the running example, the relation is essentially $loop(i, s) \rightarrow Bool$ where:

$$\begin{cases} loop(1,1) \rightarrow True \\ loop(i,s) \wedge (i * s) \leq 1000 \wedge (i+1) \leq x \implies loop(i+1, i*s) \end{cases} \tag{4.4}$$

Querying loop properties. The reason for going through the symbolization process is to

```
(declare-rel loop (Int Int))
(declare-var x Int)
(declare-var s Int)
(declare-var i Int)
(rule (loop 1 1))
(rule (let ((cond (or (> (* i s) 1000) (> (+ i 1) x))))
  (=> (and (loop i s) (not cond)) (loop (+ i 1) (* i s)))))
```

Figure 4.11: Modeling the loop in Figure 4.10 in the SMT format understandable by the muZ fixed-points engine

be able to pose loop properties as queries to SMT solvers. For example, if we would like to query whether the loop in the running example may ever return a value greater than 5000, we should post the following query to an SMT solver:

$$Query : \exists i, s, x,\ loop(i, s) \wedge (i * s) > 5000 \wedge (i + 1) \leq x \tag{4.5}$$

The answer will be `satisfiable` with $i = 7, x = 8, s = 840$ with an exact working of each loop iteration:

$$loop(1, 1) \wedge loop(2, 1) \wedge loop(3, 2) \wedge loop(4, 6) \wedge loop(5, 24) \wedge loop(6, 120) \wedge loop(7, 840)$$

4.8.4 The Object-Chunk Symbolic Memory Model

The problems caused by symbolic pointers

Symbolic memory pointers are another major source of limitations on applying symbolic execution to large programs. Suppose we encounter a statement like `int val = *ptr` during symbolization, two issues we need to resolve before we could assign any representation to the variable `val`:

1. which object(s) `ptr` might point to?
2. what is the value currently alive at that memory location?

```
// continued from Figure 5.1
// e = phi(5, 10, 15)
void *p = malloc(e);
```

(a) Allocation with a symbolic size

```
// continued from the above
void *q = malloc(f);
void *r = (e == f) ? p : q
```

(b) Multiple entries in points-to sets

```
char *g = malloc(128);
memset(g, 42, 20);
memset(g, 66, x);
```

(c) `memset` with a symbolic length

```
char *h = malloc(8);
h[0:7] = 1;

if (x > 100) {
  h[0:5] = 2;
} else if (x < 100) {
h[2:7] = 3;
} else {
  h[3:4] = 4;
}

if (x >= 100) {
h[1:6] = 5;
}
```

(d) Memory content partial override with conditions

Figure 4.12: Illustration of various challenges caused by symbolic memory

Unfortunately, neither question can be answered easily in the symbolic world. Figure 4.12 show four cases that pose challenges to existing symbolic executors.

Allocation with a symbolic size. As shown in Figure 4.12 subfigure-a, in many cases, we observe memory allocation with a symbolic size which itself can take multiple values depending on the guarding conditions. In this case, there are essentially three possibilities (5, 10, and 15) for the size of object `p`, depending on the condition of x (see Table 4.3 for details).

Multiple entries in points-to sets. Continued from the abovementioned example, there could be multiple possibilities of the pointer itself. In Figure 4.12 subfigure-b, the pointer `r` could points to either object `p` or `q` depending on the relationship between variables `e` and `f`. As a result, if we see a statement `*r = 10`, we need to account for the fact that either `p` or `q` could be changed, but not both.

Memset with a symbolic length. As shown in Figure 4.12 subfigure-b, we could change the content of multiple memory locations at once with operations like `memset` or `memcpy`. Similar to loops, if we know how many bytes are changed statically, we this could be handled easily with a chain of memory writes, like unrolling a bounded loop. The difficulty lies in unbounded range operations. In this case, how many bytes are flipped to 66 in object `g` depends on the value of x and how many bytes might remain 42 depends on the relationship between x and 20.

Memory content partial override with conditions. Even without the complexity of symbolic length, the liveliness of the memory content can still pose an issue in conjunction with the if-else branching in the control-flow. As shown in Figure 4.12 subfigure-d, the content of the 8-byte object `h` depends on the path executed in the CFG, which is further decided by the value of x. Not only that, the memory content of `h` can be different at different timestamps. For example, if we load `h[1:6]` before line 12 and after line 13, the content will be totally different as after line 13, the object is overridden by the latest store, but only partially, as the value of `h[0]` and `h[7]` are not affected. This means that KSA needs to track the liveliness of memory content throughout time.

Memory object representations

The solution proposed in KSA to solve all these problems is the object-chunk memory model. As an illustration, how the memory allocation in Figure 4.12 subfigure-a is symbolized in Table 4.4. Essentially, whenever we encountered a memory allocation with a symbolic size with N instances, we will create corresponding N objects, where each object bearing a particular size value. In this example, the symbolic size e has three guarded representations, therefore we create 3 different objects and store them in the object table with corresponding conditions. Note that we even assign different pointer values to the pointer `p` to denote that they are essentially different objects, exploiting the chance that a pointer is just a symbol of memory address and thus, we could assign different values to it according to our will.

For each object in the object table, we will further create a chunk table to hold the records of writes (*i.e.*, stores) to the actual memory content, as shown in subfigures-b,c,d. Each record in the chunk table represent on write operations with recording on the

- starting address (*i.e.*, offset),
- length,
- value stored into the bytes,

- the condition associated with the write,
- the current content (*i.e.*, the blob),
- and most importantly, the liveliness condition.

Suppose that after the allocation of object `p`, we further issue two writes to it, `p[2] = 4` and `p[2] = 0`. This will yield two records in each chunk table. And note that the liveliness condition of the first record is $True \xrightarrow{2} False$. This means that the liveliness is initially $True$ and later becomes $False$ after the overwrite at record 2. As a result, if we were to load the value of `p[2]` before the second write, we get `4`, otherwise, we get `0`.

Memory content liveliness tracking

Tracking memory content liveliness is a programmatic procedure as summarized in algorithm 6. Note that the algorithm shows the most complicated case: `memcpy(dst, src, len)`, where `dst`, `src`, `len` are all symbolic values bearing multiple instances. Symbolically, all other cases of assigning values to memory content are simplified version of `memcpy`. For example, `memset` is simplified as everything from `dst` is fixed while `*p = v` means `len` is a constant.

In general, whenever we observe a new write `op` in the object, we will go through each existing record `r` in the chunk table for that object and if `r.cond` $\wedge$ `op.cond` is satisfiable, enter a new record in the chunk table, with blob as:

`store(r.blob, op.offset, op.length, op.value).`

At the same time, update the liveliness condition for `r` to be:

`r.live = r.cond` $\wedge \neg$ `op.cond.`

Applying the algorithm on the example in Figure 4.12 subfigure-d, we can get the guarded symbolic representation for the operations on memory object `h` in Table 4.5.

Table 4.4: Illustration of the object-chunk memory model for the example in Figure 4.12 subfigure-a

(a) The object table

#	Pointer	Size	Cond.
1	`0x0001_0000`	5	$x = 0$
2	`0x0002_0000`	10	$x \leq 2$
3	`0x0003_0000`	15	$x \neq 0 \land x < 2$

(b) The chunk table for object #1, with base condition $x = 0$

#	Off.	Len.	Value	Cond.	Blob	Live
1	2	1	4	$True$	`store(2, 4)`	$True \xrightarrow{2} False$
2	2	1	0	$True$	`store(2, 0)`	$True$

(c) The chunk table for object #2, with base condition $x \geq 2$

#	Off.	Len.	Value	Cond.	Blob	Live
1	2	1	4	$True$	`store(2, 4)`	$True \xrightarrow{2} False$
2	2	1	0	$True$	`store(2, 0)`	$True$

(d) The chunk table for object #3, with base condition $x \neq 0 \land x < 2$

#	Off.	Len.	Value	Cond.	Blob	Live
1	2	1	4	$True$	`store(2, 4)`	$True \xrightarrow{2} False$
2	2	1	0	$True$	`store(2, 0)`	$True$

Algorithm 6: Memory liveliness tracking algorithm

```
   In  : $object^{src}$, $offset^{src}$, $object^{dst}$, $offset^{dst}$, $length$ - guarded symbolic
         representation (i.e., with multiple instances) of the memory objects, offsets,
         and length in memcpy(src, dst, length), respectively
   In  : $cond_{path}$ - path condition for the current basic block
1  for each instance combination of < $obj_i^{src}$, $off_i^{src}$, $obj_i^{dst}$, $off_i^{dst}$, $len_i$ > in symbolic
     values < $object^{src}$, $offset^{src}$, $object^{dst}$, $offset^{src}$, $length$ > do
2      $cond \leftarrow cond_{path}$
3          $\wedge$ $obj_i^{src}$.cond $\wedge$ $off_i^{src}$.cond
4          $\wedge$ $obj_i^{dst}$.cond $\wedge$ $off_i^{dst}$.cond
5          $\wedge$ $len_i$.cond
6      if $cond$ is satisfiable then
7          for each record $rec^{src}$ in $obj_i^{src}$.chunk_table do
8              for each record $rec^{dst}$ in $obj_i^{dst}$.chunk_table do
9                  $cond_{write} \leftarrow cond \wedge rec^{src}$.live $\wedge$ $rec^{dst}$.live
10                 if $cond_{write}$ is satisfiable then
11                     $blob_{write} \leftarrow$ ite(
12                         $off_i^{dst} \leq \lambda < off_i^{dst} + len_i$,
13                         select( $rec^{src}$.blob, $off_i^{src} + \lambda - off_i^{dst}$ ),
14                         select( $rec^{dst}$.blob, $\lambda$ )
15                     )
16                     $obj_i^{dst}$.chunk_table.add < $off_i^{dst}$, $len_i$, $blob_{write}$, $cond_{write}$ >
17                     $rec_i^{dst}$.live $\leftarrow rec_i^{dst}$.live $\wedge\neg$ $cond_{write}$
18                 end
19             end
20         end
21     end
22 end
```

Table 4.5: Application of the memory liveliness tracking algorithm on Figure 4.12 subfigure-d

#	Off.	Len.	Value	Cond.	Blob	Live
1	0	8	1	$True$	`11111111`	$True \xrightarrow{2} x \leq 100 \xrightarrow{3} x = 100 \xrightarrow{4} False$
2	0	6	2	$x > 100$	`22222211`	$x > 100 \xrightarrow{5} False$
3	2	6	3	$x < 100$	`11333333`	$x < 100$
4	3	2	4	$x = 100$	`11144111`	$x = 100 \xrightarrow{6} False$
5	1	6	5	$x > 100$	`25555551`	$x > 100$
6	1	6	5	$x = 100$	`15555551`	$x = 100$

Table 4.6: Guarded symbolic representation of versioned variables in Figure 4.13

(a) Guarded representation of `v1`

v1	**condition**
0	$\neg$ `T2(S)` $\rightarrow$ `T1(L)`
2	`T2(S)` $\rightarrow$ `T1(L)`

(b) Guarded representation of `v2`

v2	**condition**
0	$\neg$ `T1(S)` $\rightarrow$ `T2(L)`
1	`T1(S)` $\rightarrow$ `T2(L)`

(c) Guarded representation of `v3`

v3	**condition**
1	`T2(S)` $\rightarrow$ `T1(S)`
2	`T1(S)` $\rightarrow$ `T2(S)`

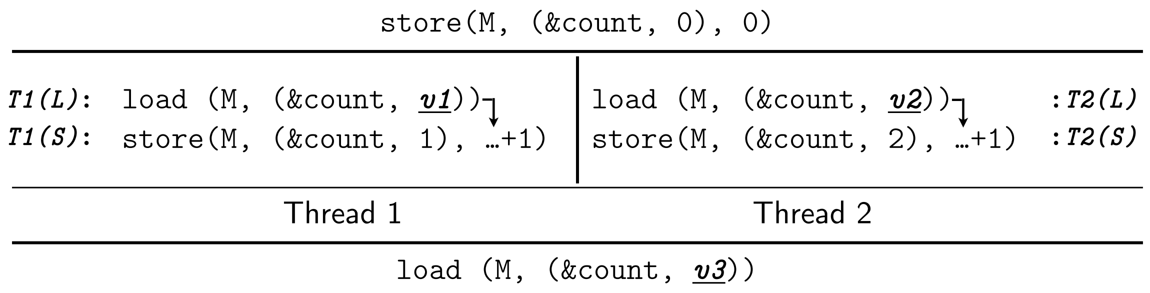
4.8.5 Concurrent Memory Accesses and Scheduling

Version tracking for memory accesses

Entering the concurrency dimension for symbolic execution, a major difference, as explored in subsection 4.4.3, is that loading from a memory address p may not always yield the same value that is stored to p *by the current thread*. Mathematically, the conventional memory model, $select(store(M, p, x), p) = x$ does not hold anymore. The reason is that after the last $store(M, p, x)$ from one thread, another thread might issue a $store(M, p, y)$ to the same memory location, which means the next load on p might get either x or y depending on which $store$ is before the load.

To account for this, in KSA, we use an additional version number to indicate that the memory content might have been overwritten by another thread since last store. Mathematically, we represent a store as $store(M, \{p, v\}, x)$ and now the new primitive on memory select-store relations $select(store(M, \{p, v\}, x), \{p, v\}) = x$ hold.

As a concrete example, Figure 4.13 shows the symbolization of the classical counter example (Figure 1.1). Whenever a thread try to load the address `&count`, it loads with a



```
                    store(M, (&count, 0), 0)
T1(L): load (M, (&count, v1))            | load (M, (&count, v2))            :T2(L)
T1(S): store(M, (&count, 1), …+1)        | store(M, (&count, 2), …+1)        :T2(S)
              Thread 1                            Thread 2
                    load (M, (&count, v3))
```

Figure 4.13: Symbolic representation of the example in Figure 1.1 (the race version) with the for-loop unrolled once and the ++ operator expanded. Timestamp variables like T1(L) represent the logic clock time when the instruction is executed.

Table 4.7: Guarded symbolic representation of versioned variables in Figure 4.14

(a) Guarded representation of v1

v1	condition
0	T1(M) $\rightarrow$ T2(M)
2	T2(M) $\rightarrow$ T1(M)

(b) Guarded representation of v2

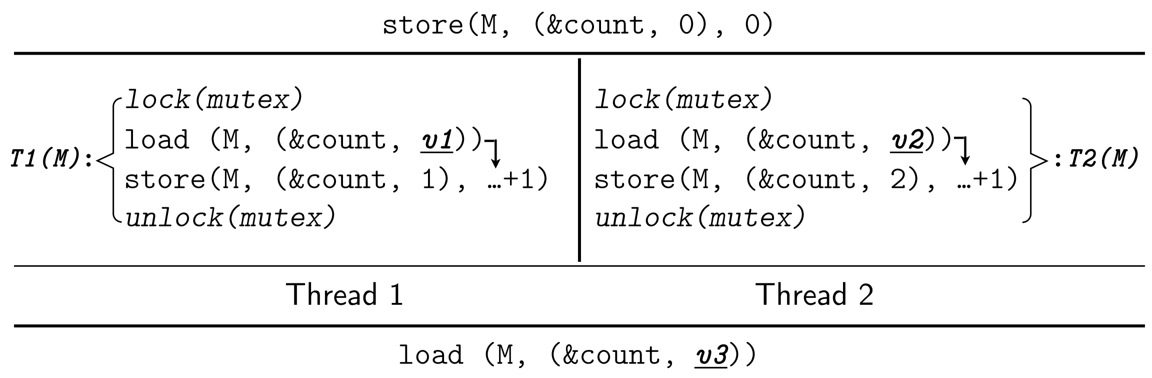
v2	condition
0	T2(M) $\rightarrow$ T1(M)
1	T1(M) $\rightarrow$ T2(M)

(c) Guarded representation of v3

v3	condition
1	T2(M) $\rightarrow$ T1(M)
2	T1(M) $\rightarrow$ T2(M)

version number, *i.e.*, v1, v2, or v3. And since there are three stores to the address &count, v1, v2, or v3 can only take values of 0, 1, or 2. Further filtered by the program order (*i.e.*, the order on the execution sequence), v1 can only take values of 0 or 2 but not 1, because assuming sequential consistency, there is no way for the load in thread 1 to see the store in thread 1 before execution (although weak memory consistency models and speculative execution might invalid this assumption, they are out of scope for KSA). Similarly, v2 can only take values of 0 or 1 but not 2, and v3 can either be 1 or 2 but never 0.

Furthermore, beyond which values the version numbers can take, we could also collect the condition in order for that particular value to materialize. For example, if v1 is 1, it means that T2(S) $\rightarrow$ T1(L), which represents that the store in thread 2 has to happen before the load in thread 1 takes place. On the other hand, if v1 is 0, it means that the store in thread 2 has not happened yet, this is denoted by neg T2(S) $\rightarrow$ T1(L). The same reasoning can be applied to all v1, v2, and v3 and the results are summarized in Table 4.6.



```
                    store(M, (&count, 0), 0)
T1(M): lock(mutex)                        | lock(mutex)
       load (M, (&count, v1))             | load (M, (&count, v2))
       store(M, (&count, 1), …+1)         | store(M, (&count, 2), …+1)   :T2(M)
       unlock(mutex)                      | unlock(mutex)
              Thread 1                    |        Thread 2
                    load (M, (&count, v3))
```

Figure 4.14: Symbolic representation of the example in Figure 1.1 (the lock version) with the for-loop unrolled once and the ++ operator expanded. Timestamp variables like `T1(M)` represent the logic clock time when the mutex lock is acquired.

Enforcing constraints on thread scheduling

After symbolizing the multi-threaded program, we want to derive a thread scheduling that leads the execution to certain states by solving constraints. Taking the example from Figure 4.13, we could check properties such as: is there a thread execution schedule that leads to `load(M, (&count, v3)) == 1`? By giving all the constraints collected in Table 4.6 to an SMT solver, we get a trace: `T1(L)` $\rightarrow$ `T2(L)` $\rightarrow$ `T2(S)` $\rightarrow$ `T1(S)` that satisfies all the constraints, which can be directly translated to an interleaving of the two threads.

To further illustrate how KSA models synchronization primitives, we also models the locked version in Figure 4.14. Compared with the racy version, a major difference is the modeling of critical sections. In the locked version, since the `mutex` lock guards a critical section, we only need to have one timestamp, `T(M)`, for the whole critical section, which basically decides who enters into the critical section first and how second. The values `v1`, `v2`, and `v3` might take do not change but their guarding conditions changed, as shown in Table 4.7. With such a modeling, we could pose the same query to the SMT solver, *i.e.*, is there a thread execution schedule that leads to `load(M, (&count, v3)) == 1`? However, this time, the solver will not be able to find an ordering between `T1(M)` and `T2(M)` to meet all constraints, and hence, we can conclude that there is no way to have `count == 1` in the locked version.

CHAPTER 5
CONCLUSION

The scale and pervasiveness of concurrent software pose challenges for security researchers: race conditions are more prevalent than ever, and the growing software complexity keeps exacerbating the situation — expanding the arms race between security practitioners and attackers beyond memory errors. As a consequence, we need a new generation of bug hunting tools that not only scale well with increasingly larger codebases but also catch up with the growing importance of race conditions.

In this thesis, I presented two complementary race detection frameworks for OS kernels: multi-dimensional fuzz testing and symbolic checking. KRACE is an end-to-end fuzzing framework that brings the concurrency aspects into coverage-guided file system fuzzing. KRACE achieves this with three new constructs: 1) the alias coverage metric for tracking exploration progress in the concurrency dimension, 2) the algorithm for evolving and merging multi-threaded syscall sequences, and 3) a comprehensive lockset and happens-before modeling for kernel synchronization primitives.

On the symbolic execution side, I presented DEADLINE for *double-fetch bugs* detection. Detecting *double-fetch bugs* without a precise and formal definition has led to a lot of false alerts where manual verification has to be involved to find real *double-fetch bugs* from the haystack of *multi-reads*. At the same time, oversimplified assumptions about how a *double-fetch bug* might appear have also caused true bugs to be missed. However, based on the formal model presented in DEADLINE, *multi-read* detection can be done through scalable and efficient static program analysis techniques, while the specialized symbolic checking engine vets each *multi-read* by precisely checking whether it satisfies all the conditions in the formal definition to become a *double-fetch bug*.

In the future work, we plan to extend DEADLINE into KSA for generic symbolic execution in OS kernels with capabilities of finding bugs in the full-fledged concurrency domain. In particular, we plan to improve a conventional symbolic execution engine with four novel techniques: 1) guarded symbolic representation to solve the path explosion problem caused by if-else branching, 2) loop modeling with recursive functions to solve the problem of unbounded loops, 3) the object-chunk memory model for handling various tricky cases related to symbolic pointers, and 4) versioning on concurrent memory accesses to model and track constraints in the concurrency dimension (*e.g.*, to enforce thread scheduling).

REFERENCES

[1] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: The next generation of ext2/3 filesystem.," in *USENIX Linux Storage and Filesystem Workshop*, 2007.

[2] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," in *Proceedings of the ACM Transactions on Storage (TOS)*, 2013.

[3] Kernel.org Bugzilla, *ext4 bug entries*, https://bugzilla.kernel.org/buglist.cgi?component=ext4, 2018.

[4] ——, *Btrfs bug entries*, https://bugzilla.kernel.org/buglist.cgi?component=btrfs, 2018.

[5] J. Corbet, *Statistics for the 4.15 Kernel*, https://lwn.net/Articles/742672/, 2018.

[6] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Trans. Storage*, vol. 10, no. 1, 3:1–3:32, Jan. 2014.

[7] MITRE Corporation, *CVE-2009-1235*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1235, 2009.

[8] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically Generating Malicious Disks Using Symbolic Execution," in *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2006.

[9] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[10] N. Wilfahrt, *Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel*, https://dirtycow.ninja/, 2016.

[11] S. Khandelwal, *11-Year Old Linux Kernel Local Privilege Escalation Flaw Discovered*, http://thehackernews.com/2017/02/linux-kernel-local-root.html, 2017.

[12] MITRE, *CVE-2017-2584*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-2584, 2017.

[13] A. Konovalov, *Exploiting the Linux Kernel via Packet Sockets*, https://googleprojectzerc blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html, 2017.

[14] J. Huang, M. K. Qureshi, and K. Schwan, "An Evolutionary Study of Linux Memory Management for Fun and Profit," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Berkeley, CA, USA, Jun. 2016, pp. 465–478, ISBN: 978-1-931971-30-0.

[15] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File Systems Unfit As Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[16] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, "Understanding Manycore Scalability of File Systems," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.

[17] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[18] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "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, Oct. 2014.

[19] J. Corbet, *Unprivileged filesystem mounts, 2018 edition*, https://lwn.net/Articles/755593, 2018.

[20] MITRE Corporation, *F2FS CVE entries*, http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=f2fs, 2018.

[21] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. 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)*, New York, NY, USA: ACM, Oct. 2019, pp. 162–180, ISBN: 978-1-4503-6873-5.

[22] J. Edge, *Kernel Address Space Layout Randomization*, https://lwn.net/Articles/569635/, 2013.

[23] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[24] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P)*, Saarbrücken, Germany, Mar. 2016.

[25] K. Lu, C. Song, T. Kim, and W. Lee, "UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[26] Silicon Graphics Inc. (SGI), *(x)fstests is a filesystem testing suite*, `https://github.com/kdave/xfstests`, 2018.

[27] SGI, OSDL and Bull, *Linux Test Project*, `https://github.com/linux-test-project/ltp`, 2018.

[28] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving Integer Security for Systems with KINT," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[29] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing Drivers without Devices," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[30] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. Checker: A Soundy Analysis for Linux Kernel Drivers," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[31] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[32] Google Inc., *Syzkaller is an Unsupervised, Coverage-guided Kernel Fuzzer*, `https://github.com/google/syzkaller`, 2019.

[33] M. Zalewski, *American Fuzzy Lop (2.52b)*, `http://lcamtuf.coredump.cx/afl`, 2019.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[35] Google Inc., *honggfuzz*, http://honggfuzz.com/, 2019.

[36] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[37] Google, *OSS-Fuzz - Continuous Fuzzing for Open Source Software*, https://github.com/google/oss-fuzz, 2018.

[38] NCC Group, *AFL/QEMU Fuzzing with Full-system Emulation*, https://github.com/nccgroup/TriforceAFL, 2017.

[39] D. R. Jeong, K. Kim, B. A. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[40] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[41] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface Aware Fuzzing for Kernel Drivers," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[42] D. Jones, *Linux system call fuzzer*, https://github.com/kernelslacker/trinity, 2018.

[43] G. C. Mateusz Jurczyk, "Bochspwn: Identifying 0-days via System-wide Memory Access Pattern Analysis," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2013.

[44] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How Double-Fetch Situations Turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[45] M. Jurczyk and G. Coldwind, *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns*, https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42189.pdf, 2013.

[46] I. Institute, *Exploiting Windows Drivers: Double-fetch Race Condition Vulnerability*, http://resources.infosecinstitute.com/exploiting-

windows-drivers-double-fetch-race-condition-vulnerability, 2016.

[47] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: Automatically Generating Inputs of Death,” in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2006.

[48] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[49] P. Godefroid, M. Y. Levin, and D. Molnar, “Automated Whitebox Fuzz Testing,” in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[50] LLVM Project, *libFuzzer - a library for coverage-guided fuzz testing*, https://llvm.org/docs/LibFuzzer.html, 2018.

[51] P. Chen and H. Chen, “Angora: Efficient Fuzzing by Principled Search,” in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[52] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, “CollAFL: Path Sensitive Fuzzing,” in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[53] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[54] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[55] NCC Group, *AFL/QEMU fuzzing with full-system emulation.* https://github.com/nccgroup/TriforceAFL, 2017.

[56] MWR Labs, *Cross Platform Kernel Fuzzer Framework*, https://github.com/mwrlabs/KernelFuzzer, 2016.

[57] H. Han and S. K. Cha, “IMF: Inferred Model-based Fuzzer,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[58] NCC Group, *A linux system call fuzzer using TriforceAFL*, https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2017.

[59] MWR Labs, *macOS Kernel Fuzzer*, https://github.com/mwrlabs/OSXFuzz, 2017.

[60] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," in *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA: ACM, Mar. 2009, pp. 25–36.

[61] K. Sen, "Race Directed Random Testing of Concurrent Programs," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, Jun. 2008.

[62] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 166–177.

[63] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[64] S. Hong and M. Kim, "Effective Pattern-driven Concurrency Bug Detection for Operating Systems," *J. Syst. Softw.*, vol. 86, no. 2, pp. 377–388, Feb. 2013.

[65] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA: ACM, Oct. 2007, pp. 103–116.

[66] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, New York, NY, USA, 2007.

[67] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-race Detection for the Kernel," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Berkeley, CA, USA: USENIX Association, Oct. 2010, pp. 151–162.

[68] M. Elver, *Add Kernel Concurrency Sanitizer (KCSAN)*, https://lwn.net/Articles/802402/, 2019.

[69] J. Alglave, W. Deacon, B. Feng, D. Howells, D. Lustig, L. Maranget, P. E. McKenney, A. Parri, N. Piggin, A. Stern, A. Yokosawa, and P. Zijlstra, *Who's afraid of a big bad optimizing compiler?* https://lwn.net/Articles/793253/, 2019.

[70] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA: ACM, Mar. 2010, pp. 167–178.

[71] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-Flow Analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*, Barcelona, Spain, Mar. 2016.

[72] L. de Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[73] ——, "Z3: An Efficient SMT Solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Berlin, Heidelberg, Mar. 2008.

[74] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, ser. Lecture Notes in Computer Science, vol. 6806, Springer, Jul. 2011.

[75] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," University of California at Berkeley, Tech. Rep., Sep. 2008.

[76] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems," in *Proceedings of the 32th International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010.

[77] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[78] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Canada, Jul. 2011.

[79] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, “Path-exploration lifting: Hi-fi tests for lo-fi emulators,” in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[80] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes, “Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying,” in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[81] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, “CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems,” in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[82] R. N. Netzer and B. P. Miller, “Detecting Data Races in Parallel Program Executions,” in *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, MIT Press, 1989, pp. 109–129.

[83] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[84] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A Dynamic Data Race Detector for Multithreaded Programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[85] M. D. Bond, K. E. Coons, and K. S. McKinley, “PACER: Proportional Detection of Data Races,” in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, Jun. 2010, pp. 255–268.

[86] Z. Anderson, D. Gay, R. Ennals, and E. Brewer, “SharC: Checking Data Sharing Strategies for Multithreaded C,” in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, Jun. 2008, pp. 149–158, ISBN: 978-1-59593-860-2.

[87] E. Pozniansky and A. Schuster, “Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs,” in *Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, New York, NY, USA: ACM, Jun. 2003, pp. 179–190, ISBN: 1-58113-588-2.

[88] D. Marino, M. Musuvathi, and S. Narayanasamy, “LiteRace: Effective Sampling for Lightweight Data-race Detection,” in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, Jun. 2009.

[89] P. McKenney, *The RCU API, 2019 edition*, https://lwn.net/Articles/777036/, 2019.

[90] Y. Cai, J. Zhang, L. Cao, and J. Liu, "A Deployable Sampling Strategy for Data Race Detection," in *Proceedings of the 24nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle, WA, Nov. 2016.

[91] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09, New York, NY, USA, 2009, pp. 62–71.

[92] S. Lu, S. Park, E. Seo, and Y. 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, Mar. 2008.

[93] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, Mar. 2010.

[94] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd. New York, NY, USA: Cambridge University Press, 2016, ISBN: 1107172012, 9781107172012.

[95] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15, ISBN: 978-1-939133-07-6.

[96] K. Owens and A. Arcangeli, *Seqlock implementation in linux*, https://github.com/torvalds/linux/blob/master/include/linux/seqlock.h, 2019.

[97] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad Hoc Synchronization Considered Harmful," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[98] Google, *syzbot*, https://syzkaller.appspot.com, 2018.

[99] O. Purdila, L. A. Grijincu, and N. Tapus, "LKL: The Linux kernel library," in *Proceedings of the 9th Roedunet International Conference (RoEduNet)*, IEEE, 2010.

[100] D. A. Ramos and D. Engler, “Under-Constrained Symbolic Execution: Correctness Checking for Real Code,” in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[101] B. Niu and G. Tan, “Modular Control-Flow Integrity,” in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[102] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Úlfar Erlingsson, L. Lozano, and G. Pike, “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM,” in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[103] J. McCarthy and J. Painter, “Correctness of a compiler for arithmetic expressions,” *Mathematical Aspects of Computer Science*, vol. 1, 1967.

[104] L. de Moura and N. Bjorner, “Generalized, Efficient Array Decision Procedures,” Microsoft Research, Tech. Rep., Sep. 2009.

[105] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, “Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features,” *ArXiv e-prints*, Nov. 2017. eprint: `1711.01254`.

[106] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing Symbolic Execution with Veritesting,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014.

[107] K. Sen, G. Necula, L. Gong, and W. Choi, “MultiSE : Multi-Path Symbolic Execution using Value Summaries,” in *Proceedings of the 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Bergamo, Italy, Aug. 2015.

[108] The LLVM Project, *LLVM Loop Terminology (and Canonical Forms)*, `https://llvm.org/docs/LoopTerminology.html`, 2020.