Efficient Race Detection in the Presence of Programmatic Event Loops

Anirudh Santhiar Shalini Kaleeswaran Aditya Kanade
Indian Institute of Science, India
{anirudh_s,shalinik,kanade}@csa.iisc.ernet.in

ABSTRACT

An event loop is the basic scheduling mechanism for programs that respond to asynchronous events. In some frameworks, only the runtime can spin event loops, while in others, these can also be spun programmatically by event handlers. The latter provides more flexibility and helps improve responsiveness in cases where an event handler must wait for some input, for example, from the user or network. It can do so while spinning an event loop.

In this paper, we consider the scheduling scheme of programmatic event loops. Programs which follow this scheme are prone to interference between a handler that is spinning an event loop and another handler that runs inside the loop. We present a happens-before based race detection technique for such programs. We exploit the structure and semantics of executions of these programs to design a sparse representation of the happens-before relation. It relates only a few pairs of operations explicitly in such a way that the ordering between any pair of operations can be inferred from the sparse representation in constant time.

We have implemented our technique in an offline race detector for C/C++ programs, called SparseRacer. We discovered 13 new and harmful race conditions in 9 open-source applications using SparseRacer. So far, developers have confirmed 8 as valid bugs, and have fixed 3. These bugs arise from unintended interference due to programmatic event loops. Our sparse representation improved efficiency and gave an average speedup of 5x in race detection time.

CCS Concepts

•**Software and its engineering** → **Dynamic analysis;** *Concurrent programming structures; Software testing and debugging;*

Keywords

Race detection, Happens-before reasoning, Event driven programs

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...
http://dx.doi.org/10.1145/2931037.2931068

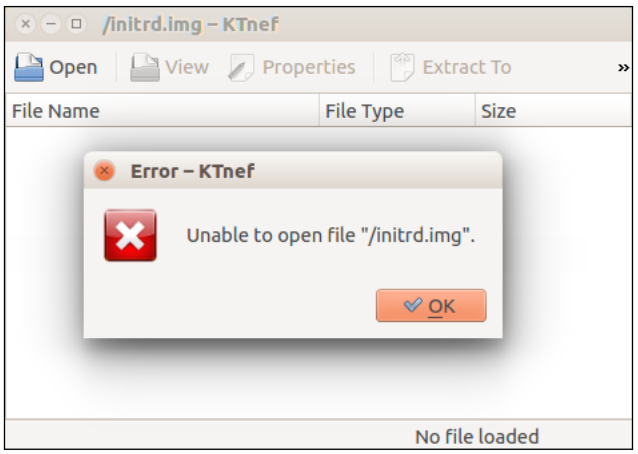

Figure 1: KTnef spins an event loop programmatically when an "Error" dialog is shown.

1. INTRODUCTION

An event loop is the basic scheduling mechanism for programs that respond to asynchronous events. The events received by a program are stored in an event queue and an event loop runs their handlers one-by-one. This is a very popular mechanism for the design and implementation of responsive applications (*e.g.*, [32, 31, 40, 27]).

Due to the non-determinism in the order of arrival of events, these programs may contain bugs that manifest only under certain event sequences. Recently, it was demonstrated that JavaScript programs [33, 37] and Android applications [19, 25] are susceptible to race conditions in which two event handlers with interfering memory operations can be reordered.

The existing race detection techniques for event-driven programs [33, 37, 19, 25, 15] assume that an event handler cannot spin event loops programmatically – only the runtime spins the (default) event loop. While this assumption holds for JavaScript and Android programs, many event-driven frameworks such as GUI libraries [1, 2, 3], web browsers [4, 5] and standard OS APIs [6, 7] also support the more flexible mechanism of programmatic event loops. In some cases, a handler may have to wait for an input, for example, from the user or network. Programmatic event loops help in improving responsiveness in such cases. Instead of blocking the thread, the handler pauses itself and spins an event loop programmatically. The event loop processes the pending events until it encounters the input event the paused handler is waiting for. Once the loop terminates, the paused handler

```
1  void KTNEFMain::loadFile(...)
2  {
3    ...
4    if ( !mParser->openFile( filename ) )
5    {
6       // The following line results in an eventual
            call to QEventLoop::exec
7       KMessageBox::error(...);
8       // mView is a field of KTNEFMain
9       this->mView->setAttachments(...);
10      ...
11   }
12   ...
13 }
14 int QEventLoop::exec(...){
15   ...              // PAUSE
16   while (!exit)
17     // programmatic event loop
18     processEvents(...);
19   ...              // RESUME
20 }
```

accesses freed object `this`

Figure 2: Code fragments of KTnef showing a programmatic event loop (lines 16–18) and an access susceptible to use-after-free (line 9).

resumes its computation.

In this paper, we consider the combined scheduling scheme of the default event loop and programmatic event loops. Programs designed using this scheme ensure responsiveness but are prone to interference between the paused handler and the handlers running in its programmatic event loop. We present a happens-before [23] based race detection technique for such programs and use it to find harmful race conditions in open-source applications, several of which are already confirmed as bugs by the developers.

We motivate our technique through a bug discovered by our tool in a Linux-based e-mail attachment viewer and extractor, KTnef [8]. This bug causes a use-after-free crash when a specific event is processed within a programmatic event loop. In the buggy scenario, the user opens a file with an unsupported format, and the application displays an error dialog as shown in Figure 1. Trying to open the file results in invocation of the method `loadFile` shown in Figure 2. It is an instance method with the implicit `this` parameter for the receiver object. If the file fails to parse (line 4), the application displays an error dialog (line 7) and awaits further input from the user. Only after the user clicks the "OK" button, it can proceed to the main window.

The method `MessageBox::error` called at line 7 eventually calls `QEventLoop::exec`, which spins an event loop programmatically at lines 16–18. The method `processEvents` processes the next pending event. When the click event for the "OK" button is handled in this loop, its handler sets the `exit` flag, causing the loop to terminate (line 16). While the loop is running, we can schedule an event whose handler H frees the object `this` on which `loadFile` was invoked. After termination of the event loop, the control returns to line 9 that dereferences the freed object `this` and results in a crash. We reported this bug to the developers. They have confirmed it to be a valid bug and have also issued a fix for it [9].

Detecting this bug was difficult because in the test runs, we could observe the handler H (which frees the object) only in the default event loop *after* the "Error" dialog had finished. Our race detector discovered that there was no happens-before ordering to prevent scheduling of H within the programmatic event loop. The existing race detection approaches [33, 37, 19, 25, 15] model only the default event loop in which handlers run to completion and hence, cannot identify that H can be scheduled while the handler that opens the "Error" dialog is paused to spin an event loop. Since this is the only scenario where the bug manifests, they would fail to detect this bug.

To detect races in programs that use programmatic event loops, we have designed a *trace language* to record the event scheduling operations and memory accesses, and a set of *happens-before rules* to detect possible reorderings of these operations. We exploit the structure and semantics of executions of event-driven programs to design *a sparse representation of the happens-before relation*. It relates only a few pairs of operations explicitly in such a way that the ordering between any pair of operations can be inferred from the sparse representation in constant time. We use the sparse representation to improve efficiency of race detection.

We have implemented our technique in an offline race detector for C/C++ programs, called SparseRacer. We have implemented a trace generation mechanism to obtain execution traces of programs. A generated trace is analyzed by SparseRacer for race conditions by computing a sparse representation of the happens-before relation.

We have applied SparseRacer on several open-source applications to detect races between use and free operations, called *use-free* races [19]. These races, when exercised, invariably cause crashes leading to loss of application state. Use-after-free situations are also known to be exploitable security vulnerabilities [14]. We found 13 new use-free races in 9 open-source applications including the bug in KTnef discussed earlier and bugs in other popular applications such as the document viewer Okular [10]. We reported all of these to the developers, and they have confirmed 8 of them to be valid bugs, and have fixed the bugs in KTnef, Kate and KWrite. These bugs arise from unintended interference due to programmatic event loops.

The existing race detection techniques for event-driven programs [33, 37, 19, 25, 15] cannot identify them due to their inability to reason about programmatic event loops. During testing, we only observed the free operations after the use operations. Therefore, the memory safety tools such as Valgrind [29] and AddressSanitizer [39] also could not detect them. Our sparse representation resulted in an average speedup of 5x in race detection time. Due to the likely severity of use-free races, we have focused on them in this work. SparseRacer can be readily applied to detect the usual data races between read-write operations also.

- This paper analyzes the programmatic event loop mechanism used extensively in event-driven programs and provides a trace language to capture their asynchronous event-handling behavior.
- It presents a set of happens-before rules to detect race conditions in programs that use programmatic event loops along with the default event loop. The existing approaches handle only the latter.
- It presents a sparse representation for efficient computation of the happens-before relation.
- The techniques are implemented in a race detector SparseRacer and used for detecting 13 new and harmful race conditions in open-source applications, 8 of which are already confirmed as bugs by the developers.

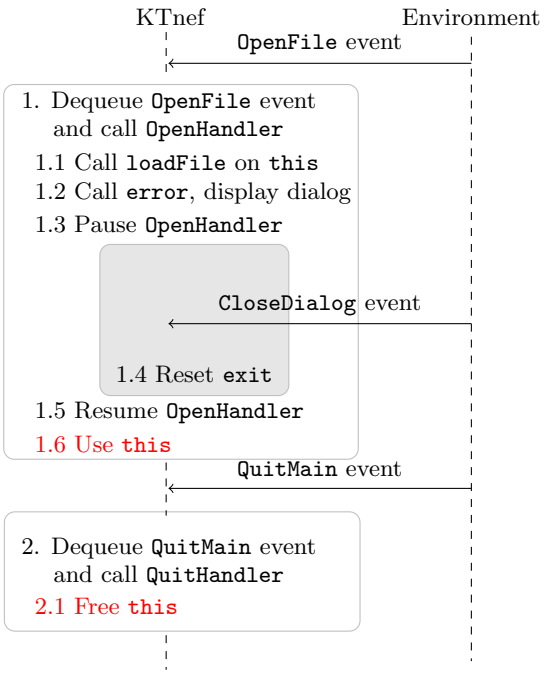

Figure 3: An execution of the KTnef application. The shaded rectangle shows the time-span during which the programmatic event loop executes.

2. DETAILED EXAMPLE

We show a representative execution trace of the KTnef application in Figure 3. This execution does not exhibit the use-after-free situation. We illustrate how SparseRacer identifies the use-free race discussed in the Introduction by analyzing this execution trace.

During this execution, the user triggers the `OpenFile` event to open a selected file. In response, KTnef invokes the handler called `OpenHandler` in step 1 that calls the method `loadFile` shown in Figure 2. In step 1.2, it displays the error dialog shown in Figure 1 provided the file fails to parse.

The application waits for the user to acknowledge the error by clicking "OK". Instead of waiting idly and blocking the thread, the application can respond to other pending events. For this, it pauses the current handler (step 1.3) and spins an event loop programmatically. Figure 4 shows the call stack of the paused handler. The event loop calls the handlers of pending events synchronously one-by-one through the `processEvents` method in Figure 2.

When the user clicks the "OK" button, an event called `CloseDialog` is sent to the application. The handler of this event runs in the programmatic event loop. It resets the `exit` flag (step 1.4). In the next iteration of the loop in the `exec` method, since the `exit` flag is true, the event loop terminates and the paused handler gets the control. This is marked by the Resume operation (step 1.5) in Figure 3. The dereference of the `this` object is step 1.6. Suppose the user then triggers the `QuitMain` event by navigating to the file menu of the main window of KTnef, and clicking close. The handler of this event frees the `this` object on which `loadFile` was invoked earlier. The object is not accessed subsequently, and hence, this execution does not contain a use-after-free. However, the use may follow the free under a different event ordering.

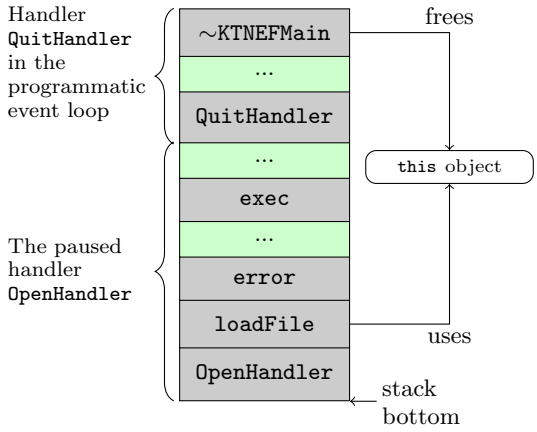

Figure 4: Call stack showing interference between a paused handler and a handler running inside the event loop spun by the paused handler.

The `OpenFile` and `QuitMain` events are co-enabled. The user can select any of these non-deterministically and there is no ordering between the two. Suppose the `CloseDialog` event is related to `QuitMain` by the happens-before relation. In other words, `QuitMain` cannot be triggered before `CloseDialog`. As the handler of `CloseDialog` signals termination of the programmatic event loop by resetting the flag `exit`, the handler of `QuitMain` cannot run within the event loop and the free operation cannot interfere with the use operation. However, there is no ordering between `CloseDialog` and `QuitMain` events and the latter can be processed within the programmatic event loop if triggered before `CloseDialog`. As shown in Figure 4, the paused handler `OpenHandler` holds a reference to the `this` object, and the handler of `QuitMain`, which frees it, is run inside the programmatic event loop. The paused handler ends up accessing the freed object when resumed, resulting in a crash. If `QuitMain` is scheduled before `OpenFile`, then the application simply exits, and there is no use-after-free.

SparseRacer finds that there is a race between the use and free operations of `OpenHandler` and `QuitHandler` given the execution trace in Figure 3. It infers that `QuitHandler` can execute after step 1.3 and before step 1.4. These steps respectively indicate when the programmatic event loop starts and when the flag to terminate it is set. Step 1.6 happens after the termination of the event loop. SparseRacer thus infers that the free operation in step 2.1 may happen before the use operation in step 1.6. Detecting this race involves happens-before reasoning [23]. In the next section, we present a set of rules to infer happens-before relation for the execution traces similar to Figure 3.

We observe that it is unnecessary to represent all happens-before orderings explicitly. Figure 5 shows two blocks of operations from the trace of Figure 3. The block B1 contains operations of `OpenHandler` from the beginning up to the operation that pauses it to spin a programmatic event loop. The block B2 contains operations of the handler of the `CloseDialog` event. Since the `CloseDialog` event can only be dequeued after `OpenHandler` pauses, all the operations of B2 happen after all the operations of B1. In a naïve representation, all these orderings will be represented explicitly. However, the operations within each block execute sequentially and no other event handler can be executed in

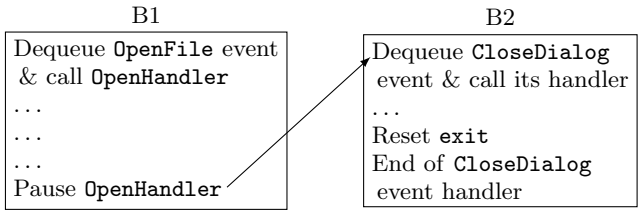

Figure 5: In the sparse representation of the happens-before relation, instead of relating every operation in B1 to every operation in B2 by happens-before ordering, we only relate the last operation of B1 to the first operation of B2.

between them. Therefore, it suffices to establish an ordering between the last operation of B1 and the first operation of B2 as shown in Figure 5. The ordering between any pair of operations (α_1, α_2) where $\alpha_1 \in$ B1 and $\alpha_2 \in$ B2 is implied by this ordering combined with the order of execution (or program order) within the individual blocks. We exploit this observation to obtain a sparse representation of the happens-before relation for event-driven programs. As we discuss in the next section, the happens-before ordering between any pair of operations can be inferred from the sparse representation in constant time.

3. TECHNICAL DETAILS

The problem of race conditions is well-studied for multi-threaded programs. Two interfering operations have a *race* if they are not ordered by the happens-before relation. As observed in [33, 37] for JavaScript programs and in [19, 25] for Android programs, event handlers running on the *same* thread are also susceptible to races due to possible reorderings of events. The above approaches assume that an event handler runs to completion on a thread.

The presence of programmatic event loops makes programs more prone to races since a handler which spins an event loop is *interleaved* with the handlers of events processed within the loop. Thereby, a handler running in a programmatic event loop can interfere with the paused handler. Further, the case of programmatic event loops becomes more challenging because the event loops can be entered *recursively*. That is, a handler H running in a programmatic event loop can spin its own event loop and so on, resulting in many possible interleavings of different handlers.

In this section, we present the technical details of our approach to detect races in handlers running on the same thread where the handlers may spin programmatic event loops. Our approach can be extended for multi-threaded programs but we leave it to future work. In this work, we only model the main thread of a program. This was sufficient to detect many real races in several open-source applications as reported in Section 5.

3.1 Trace Language

We first present a trace language to capture the asynchronous event-handling behavior of programs that use programmatic event loops along with the default event loop. Our trace language consists of the following operations:

- Memory access: Operations `read(x)` and `write(x)` state that the thread has read or written a memory location `x`. Operation `alloc(x,n)` indicates that it has allocated `n` bytes starting at memory location `x`. Operation `free(x,n)` means that it has freed `n` bytes starting at location `x`.

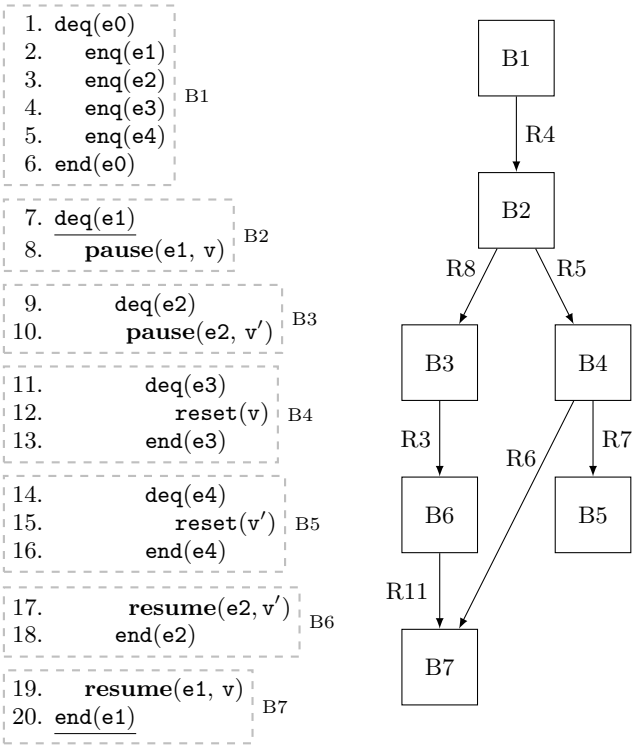

Figure 6: On the left is an example trace with the blocks identified by dashed rectangles. On the right is a subset of happens-before orderings inferred among the blocks. A directed edge between blocks B and B′ indicates that the last operation of B happens before the first operation of B′. Each edge is annotated with ID of the rule used for inferring it.

- Event dispatch and processing: Operation `enq(e)` means that the thread has enqueued an event `e` to itself. Events can carry data but we do not require it for race detection. We assume that each event has a unique ID in an execution trace. This assumption can be met by suitable renaming of events in the trace before analysis. Operation `deq(e)` states that the thread has dequeued event `e` from its event queue and called its handler. When it finishes execution of the handler of event `e`, we get the operation `end(e)`. The environment can enqueue events to the program. If we have only the `deq` operation of an event in our trace, without a corresponding `enq`, then that event is posted by the environment. To model some ordering involving environmental events, the enqueues for those events can be explicitly tracked in the execution trace.

- Default event loop: Operation `enterloop` indicates that the thread has started processing events from its event loop. Until then, it performs sequential computations, for example, to initialize some data structures. The thread stops processing the event queue by executing the `exitloop` operation.

- Programmatic event loop: Operation `pause(e,v)` indicates that the handler of event `e` is paused to spin a programmatic event loop. The loop runs until the boolean variable `v`, called the *loop guard*, is set to true by some event handler running in the programmatic event loop. The operation which sets `v` to true is represented as `reset(v)`. Operation `resume(e,v)` is the first operation of the paused handler after the event loop guarded by `v` is terminated. For simplicity, we assume that each programmatic event loop in a trace has a unique loop guard.

An *execution trace* $\rho = \langle \alpha_1, \ldots, \alpha_n \rangle$ of an event-driven program is a valid sequence of operations in our trace language. On the left side in Figure 6 is an example trace. The handler of event `e0` enqueues four events that will be processed in the order of their enqueues. The handler of event `e1` is called next. Both `e0` and `e1` are processed in the default event loop. For brevity, we omit the `enterloop` and `exitloop` operations from the trace. The handler of `e1` spins a programmatic event loop guarded by variable `v`. The loop runs between operations 8–19. The next pending event `e2` is dequeued in this event loop. Its handler spins an event loop guarded by variable $\texttt{v}'$. This is a recursive event loop. The handler of `e3` is called in this loop and it resets the guard `v` of the outer event loop spun by `e1`. The inner loop continues since $\texttt{v}'$ is not yet reset. The handler of `e4` resets it and causes termination of the inner loop at operation 17. After `e2` finishes, the event loop of `e1` terminates in operation 19 since its guard variable `v` is already set.

An event handler can run two or more event loops sequentially. We call this the case of *cascaded* loops. For simplicity, we focus the remaining discussion under the assumption that an event handler spins at most one event loop. Later in the section, we briefly discuss about cascaded loops.

3.2 Blocks of Operations

An event handler that does not spin an event loop runs to completion before the next event can be processed. Thus, its operations are not interleaved with the operations of any other event handler. When two such events are enqueued in a different order, the respective event handlers are reordered *entirely*. Even though the operations of an event handler that spins an event loop are interleaved with those of other handlers, there is a clear structure to the possible interleavings. The handler that spins an event loop is *paused* until some handler running in the loop resets the loop guard. Once the paused handler *resumes*, no other event handler can interleave with it. This structure applies recursively in the presence of recursive programmatic event loops. We exploit this observation to obtain a coarse-grained representation of an execution trace of an event-driven program.

For this, we define a *block* of operations as a maximal sequence of contiguous operations of the same event handler such that operations from another handler cannot interleave with them. Thus, a block stretches from the `deq` operation to the `end` operation for an event handler that does not spin an event loop. If it does spin an event loop then one block stretches from the `deq` operation to the `pause` operation, and another one from the `resume` operation to the `end` operation. In Figure 6, we enclose operations of the example trace belonging to the same block in a dashed rectangle. The possible interleavings of the operations of the trace containing 20 operations can now be analyzed precisely over the corresponding 7 blocks. In practice, the number of operations in a trace can be very large but the corresponding number of blocks is much smaller. As reported in Section 5, we observed a few orders of magnitude reduction from the number of operations to the number of blocks.

We shall use blocks to obtain a sparse representation of the happens-before relation. In a multi-threaded program, different threads may interleave at the granularity of individual operations and a block-level representation of the happens-before relation is not feasible. However, for multi-threaded *event-driven* programs, the overall happens-before relation can be decomposed into single-threaded and multi-threaded relations [25] where the single-threaded relation relates event handlers running on the same thread. It would be interesting to explore whether the techniques of this paper can be extended to multi-threaded event-driven programs.

3.3 Happens-before Relation

For an execution trace $\rho = \langle \alpha_1, \ldots, \alpha_n \rangle$, the *happens-before relation* $\preceq$ is a partial-order over $\{\alpha_1, \ldots, \alpha_n\}$ such that $\alpha_i \preceq \alpha_j$ implies that in any valid reordering of operations of ρ, α_i happens before α_j. Conversely, if there is no ordering between α_i and α_j then they can appear in different orders in different runs of the program.

We give a set of rules in Figure 7 to infer the happens-before relation for a trace ρ. We use $\alpha_i \in \mathrm{B}$ to indicate that operation α_i belongs to block B. Similarly, we use $\alpha_i \in \texttt{E}$ to indicate that α_i belongs to the handler of event `E` and $\mathrm{B} \in \texttt{E}$ to indicate that block B belongs to the handler of `E`. For a block B, let `first`(B) be the first operation in B and `last`(B) be the last operation. Our rules work even if programmatic event loops are spun recursively.

On the right side in Figure 6 is a graph representing some happens-before orderings for the example trace. We refer to it to illustrate the rules.

3.3.1 Inference Rules

The rules R1-R4 in Figure 7 follow immediately from the semantics of programs. The rule R1 states that except for the block to which `enterloop` belongs, the first operation of any block happens after `enterloop`. The rule R2 is analogous but for `exitloop`. The operations of an event handler are executed sequentially and hence, are ordered by their order of execution, also called the *program order*. If $\mathrm{B}, \mathrm{B}' \in \texttt{E}$ and `last`(B) appears before `first`(B') in the execution trace, then by R3, `last`(B) $\preceq$ `first`(B'). In Figure 6, the last operation of B3 appears before the first operation of B6 and both belong to the handler of event `e2`. Using R3, we can derive the edge between them. Let `enq`(`E`) $\in$ B. By definition, all operations until `last`(B) are executed sequentially. As a consequence, the handler of `E` cannot interleave between `enq`(`E`) and `last`(B). Thus, we have the ordering `last`(B) $\preceq$ `deq`(`E`) given by R4. This rule helps us derive the edge (B1, B2) in Figure 6.

The rules R5-R6 relate a paused handler with the handler that resets its loop guard. If `E1` is spinning an event loop with loop guard `v` and `reset`(`v`) $\in$ `E2` then the pause operation of `E1` happens before the dequeue operation of `E2`, as stated in R5. The rule R6 similarly relates the end operation of `E2` with the resume operation. We use these rules to infer the edges (B2, B4) and (B4, B7) in Figure 6. As stated before, our rules work for recursive loops also. For example, the handler of event `e3`, which resets the loop guard of `e1`'s loop,

$$\boxed{\text{R1}}\ \frac{\texttt{enterloop} \notin \text{B}}{\texttt{enterloop} \preceq \texttt{first}(\text{B})} \qquad \boxed{\text{R2}}\ \frac{\texttt{exitloop} \notin \text{B}}{\texttt{last}(\text{B}) \preceq \texttt{exitloop}} \qquad \boxed{\text{R3}}\ \frac{\text{B}, \text{B}' \in \texttt{E} \quad \texttt{last}(\text{B}) \text{ appears before } \texttt{first}(\text{B}')}{\texttt{last}(\text{B}) \preceq \texttt{first}(\text{B}')}$$

$$\boxed{\text{R4}}\ \frac{\texttt{enq}(\texttt{E}) \in \text{B}}{\texttt{last}(\text{B}) \preceq \texttt{deq}(\texttt{E})} \qquad \boxed{\text{R5}}\ \frac{\texttt{reset}(\texttt{v}) \in \texttt{E2}}{\texttt{pause}(\texttt{E1}, \texttt{v}) \preceq \texttt{deq}(\texttt{E2})} \qquad \boxed{\text{R6}}\ \frac{\texttt{reset}(\texttt{v}) \in \texttt{E2}}{\texttt{end}(\texttt{E2}) \preceq \texttt{resume}(\texttt{E1}, \texttt{v})}$$

$$\boxed{\text{R7}}\ \frac{\texttt{E1}\text{'s handler does not spin an event loop} \quad \texttt{enq}(\texttt{E1}) \preceq \texttt{enq}(\texttt{E2})}{\texttt{end}(\texttt{E1}) \preceq \texttt{deq}(\texttt{E2})} \qquad \boxed{\text{R8}}\ \frac{\texttt{E1}\text{'s handler spins an event loop} \quad \texttt{enq}(\texttt{E1}) \preceq \texttt{enq}(\texttt{E2})}{\texttt{pause}(\texttt{E1}, \texttt{v}) \preceq \texttt{deq}(\texttt{E2})} \qquad \boxed{\text{R9}}\ \frac{\texttt{last}(\text{B1}) \preceq \texttt{first}(\text{B2}) \quad \texttt{last}(\text{B2}) \preceq \texttt{first}(\text{B3})}{\texttt{last}(\text{B1}) \preceq \texttt{first}(\text{B3})}$$

$$\boxed{\text{R10}}\ \frac{\texttt{resume}(\texttt{E1}, \texttt{v}) \preceq \texttt{enq}(\texttt{E2})}{\texttt{end}(\texttt{E1}) \preceq \texttt{deq}(\texttt{E2})} \qquad \boxed{\text{R11}}\ \frac{\texttt{E1}\text{'s handler spins an event loop guarded by } \texttt{v} \quad \texttt{enq}(\texttt{E1}) \preceq \texttt{enq}(\texttt{E2}) \preceq \texttt{enq}(\texttt{E3}) \qquad \texttt{reset}(\texttt{v}) \in \texttt{E3}}{\texttt{end}(\texttt{E2}) \preceq \texttt{resume}(\texttt{E1}, \texttt{v})}$$

$$\boxed{\text{R12}}\ \frac{\texttt{E1}\text{'s handler spins an event loop guarded by } \texttt{v} \qquad \texttt{reset}(\texttt{v}) \in \texttt{E2} \quad \texttt{E2}\text{'s handler does not spin an event loop and runs immediately within the loop of } \texttt{E1} \qquad \texttt{enq}(\texttt{E2}) \preceq \texttt{enq}(\texttt{E3})}{\texttt{end}(\texttt{E1}) \preceq \texttt{deq}(\texttt{E3})}$$

$$\boxed{\text{R13}}\ \frac{\alpha_i, \alpha_j \in \text{B} \qquad i < j}{\alpha_i \preceq \alpha_j} \qquad \boxed{\text{R14}}\ \frac{\alpha_i \in \text{B} \qquad \alpha_j \in \text{B}' \qquad \text{B} \neq \text{B}' \qquad \texttt{last}(\text{B}) \preceq \texttt{first}(\text{B}')}{\alpha_i \preceq \alpha_j}$$

Figure 7: Happens-before rules.

is run in the recursive loop executed by `e2` but the orderings inferred by R5 and R6 still apply.

Events are dequeued from the event queue in the order of their enqueues. Thus, the rule R7 states that if `E1` is enqueued before `E2` then `E1`'s handler will end before `E2`'s begins, provided `E1`'s handler does not spin a programmatic event loop. This condition on `E1`'s handler is crucial. In Figure 6, $\texttt{enq(e2)} \preceq \texttt{enq(e3)}$. However, $\texttt{end(e2)} \not\preceq \texttt{deq(e3)}$ since `e2`'s handler runs a programmatic event loop that dequeues `e3`. If `E1`'s handler does spin an event loop then $\texttt{pause}(\texttt{E1}, \texttt{v}) \preceq \texttt{deq}(\texttt{E2})$ as stated in the rule R8. The rule R9 defines transitive ordering over blocks. Figure 6 shows examples of edges inferred using rules R7 and R8. To avoid clutter, we have not shown any transitively computed edge in the figure. If the enqueue of an event `E2` happens after the resume operation of `E1` then `E1` finishes before `E2` can be dequeued, as per the rule R10.

Suppose the handler of an event `E1` spins an event loop which is reset by the handler of another event `E3`. Let the enqueue of a third event `E2` happen in-between those of `E1` and `E3`. The rule R11 states that `E2` finishes before the resume operation of `E1`. Note that the consequent holds even if `E2` were to spin an event loop of its own. This rule allows us to infer the edge (B6, B7) in Figure 6. Let the handler of an event `E2` reset the loop guard of the event loop of `E1`. Suppose an event `E3` is enqueued after `E2`. If `E2` does not spin an event loop and runs immediately within the programmatic event loop of `E1` then the rule R12 tells us that `E1` finishes before `E3` is dequeued. The antecedent conditions of R12 on `E2` are necessary. For example, in Figure 6, `e3` resets the loop guard `v` of the event loop of `e1` but it does not run immediately within the event loop of `e1`. Even though `e4` is enqueued *after* `e3`, it finishes *before* `e1` finishes.

By definition, operations within a block are ordered by program order as stated by R13. Let $\alpha_i \in \text{B}$ and $\alpha_j \in \text{B}'$ such that $\text{B} \neq \text{B}'$. By the definition of blocks, $\alpha_i \preceq \alpha_j$ iff $\texttt{last}(\text{B}) \preceq \texttt{first}(\text{B}')$. This is encoded in the rule R14.

3.3.2 Sparse Representation

As argued before, valid reorderings of operations of a trace can be analyzed at the level of blocks without losing precision. Let $\mathcal{B}$ be the set of blocks of an execution trace ρ, and F and L be the set of first and last operations of blocks in $\mathcal{B}$. The *sparse representation* $\sqsubseteq$ *of the happens-before relation* $\preceq$ for a trace ρ is a subset of $L \times F$ such that for any $\alpha_i \in L$ and $\alpha_j \in F$, $\alpha_i \sqsubseteq \alpha_j$ iff $\alpha_i \preceq \alpha_j$. All the edges in the graph in Figure 6 belong to the sparse representation of the happens-before relation of the example trace.

A closer look at the rules in Figure 7 reveals that the rules R1-R12 only infer $\alpha_i \preceq \alpha_j$ for $\alpha_i \in L$ and $\alpha_j \in F$. By definition, these belong to the sparse representation. The rule R13 adds ordering between operations belonging to the same block (*i.e.*, intra-block orderings). The rule R14 adds ordering between operations belonging to different blocks (*i.e.*, inter-block orderings). Interestingly, we can infer them from the sparse representation in *constant time* without explicitly representing them. By assigning integer indices in an increasing order to the operations within each block, we can infer an intra-block ordering in constant time. The antecedent condition $\texttt{last}(\text{B}) \preceq \texttt{first}(\text{B}')$ of rule R14 can be checked in constant time, given the sparse representation. Thus, to infer whether $\alpha_i \preceq \alpha_j$, we can use the sparse representation $\sqsubseteq$ defined in terms of the blocks.

The sparse representation can be used for event-driven programs which do not use programmatic event loops also. As stated before, an event handler can run two or more cascaded event loops sequentially. We have extended the rules discussed above to handle cascaded loops but we skip these extensions due to limitations of space. The extensions are reasonably straightforward. For example, let the

resume operation of `E1` in the rule R10 belong to a block B. Then to make the rule apply even if `E1` were to spin a cascaded loop following the resume, we can make the consequent `last`(B) $\preceq$ `deq`(`E2`). If `E1` does not spin another loop sequentially, `last`(B) will be the end operation of `E1` (as considered in R10) else it will be the pause operation of the next loop spun by `E1`.

3.3.3 Computing the Sparse Representation

To compute the sparse relation $\sqsubseteq$, we initialize it to an empty set and apply the rules R1-R12 in Figure 7 repeatedly until it reaches a fixpoint. The rules R13 and R14 are used only as *queries* to resolve some orderings used in the antecedents of the other rules but the orderings derived from them are not stored in $\sqsubseteq$.

In our implementation, we have used a directed graph similar to the one in Figure 6 as a data-structure to compute the sparse representation. Vector clocks [26] can provide a more efficient alternative. Vector clocks can be defined for event-driven programs by assigning one component of a vector clock VC to each event handler [37]. The chain decomposition optimization [21] is used in [37] to reduce the number of clocks in a vector clock representation: Instead of assigning a clock component to each event handler separately, it can be assigned to a chain of handlers having a total order among them. In our case, we can achieve a similar optimization by assigning a clock component to a chain of blocks. It is beyond the scope of this paper to elaborate more on this. We intend to discuss and evaluate it in future work.

4. IMPLEMENTATION

We have implemented a trace generation mechanism for C/C++ programs and a race detector, called SparseRacer. SparseRacer analyzes the generated traces for race conditions using the sparse representation of the happens-before relation. We now give details of our implementation.

4.1 Trace Generation

We instrument programs to generate execution traces in our trace language. Unlike the standard POSIX APIs for multi-threading [11], presently, there are no standard APIs for event-driven frameworks. We therefore have to instrument the event processing code in each framework separately to trace event processing operations. We simplify this by designing a small library which emits primitives in our trace language and place calls to the library functions at appropriate locations in the framework code manually.

For the purposes of experimentation, we consider a popular cross-platform application framework Qt [12] and analyze applications written using Qt for race conditions. Qt supports programmatic event loops. The entire Qt framework comprises 2.5M source lines of C++ code but to trace event processing operations, we had to add less than 100 lines of code to it manually. We inserted calls to our library in the `QEventLoop` class, referred to in the Introduction, and call sites to its `exec` method. The enqueue and dequeue operations are part of methods of other classes. To track the enqueue operations, we added code to the `addEvent` method. The dequeue operations are tracked by adding code to the `sendPostedEvents` and `processEvents` methods. In Qt, events are assigned discrete priority values when they are enqueued. Our instrumentation tracks these priorities, and we apply rules R7, R8, R11 and R12 from Figure 7 only when the enqueues in the antecedents, in addition to being related by $\preceq$, are of events with the same priority.

We wrote LLVM Clang [24] compiler passes to record memory operations and debug information. For analysis, we chose Qt-based applications developed for version 4.14.4 of the KDE graphical desktop environment [13] for Linux. Memory accessed by application code was often allocated and freed by the KDE framework. To track memory operations accurately, we instrumented both the application code and the entire KDE code base using our LLVM passes. In a post-processing pass, we discard allocation and free operations for memory not accessed within the application's main thread.

The applications we analyzed and the underlying frameworks, Qt and KDE, are big and complex. It would be quite challenging to triage races without adequate debugging information. Therefore, we wrote LLVM passes to record debugging information. In particular, the instrumented application binaries log function entry and exit, so that stack traces at interesting operations like `alloc`, `free`, `read`, `enq`, etc. are available. If α_i enqueues an event whose handler executes another enqueue operation α_j then we refer to α_i as the *parent* of α_j and to α_j as the *child* of α_i. As a debugging aid, the instrumentation also emits all sequences of enqueue operations such that the two successive operations are related by the parent-child relation.

4.2 Race Detection

SparseRacer takes as input an execution trace of an event-driven program and computes the sparse representation of the happens-before relation as discussed in Section 3.3.3. It identifies the blocks in a single pass over the entire trace. If it finds a pair of interfering operations that do not have a happens-before ordering between them, it treats them as a potential race. Due to the likely severity of use-free races, we applied SparseRacer to identify races between use and free operations on the same memory location. A prior work [19] observed that a use-free race is *benign* if the handler X that performs the use contains 1) a matching allocation before the use and 2) it runs on the same thread as the handler Y that performs the free. In the presence of programmatic event loops, we need one more condition: 3) the handler X should not run a programmatic event loop. We implemented this check in SparseRacer to filter out benign use-free races. If SparseRacer finds a non-benign use-free race then it reports the corresponding event handlers as *interfering*.

Qt provides *deferred delete* events to asynchronously schedule objects for deletion. If a race report involved a free operation belonging to the handler of a deferred delete event, and a use operation in a handler that did not spin a programmatic loop, we filtered it out. This was done to faithfully model Qt's semantics, where the latter cannot be reordered to execute after a deferred delete event. Even though we have used SparseRacer to report use-free races, it can be applied readily to identify data races over read-write operations also.

5. EXPERIMENTAL EVALUATION

We performed experiments to evaluate 1) whether race conditions can arise in practice due to unintended interleavings in programmatic event loops and 2) whether SparseRacer can detect them efficiently. Using SparseRacer, we did find several harmful race conditions between paused event handlers and handlers running inside their programmatic event loops. We now discuss our experiments in detail.

Table 1: Applications analyzed for race conditions

Application	Description
Ark	File compression/decompression utility
Cervisia	Version control front end for CVS
Kate	Advanced text editor
KDF	File system utility
Kolourpaint	Raster graphics editor
KOrganizer	Scheduling and calendar app
KTnef	E-mail attachment viewer/extractor
KWrite	Lightweight text editor
Okular	Universal document viewer

Table 2: Summary of experimental results

Application	#Ops	#Evts	#Blks	#Interf. Handlers		Time in sec
				Total	True	
Ark	5008	127	148	6	2	2
Cervisia	1726	129	156	14	2	0.3
Kate	83633	194	225	4	1	480
KDF	1089	15	24	1	1	0.2
Kolourpaint	12746	67	75	2	1	12
KOrganizer	58232	273	290	12	2	179
KTnef	1158	258	275	1	1	0.3
KWrite	74105	62	75	4	1	396
Okular	16785	223	273	14	2	15
Total				58	13	

5.1 Experimental Setup

We selected several Qt-based applications running on KDE for analysis. Table 1 gives the list of nine applications we analyzed for race conditions. These include the KTnef e-mail attachment viewer discussed in the Introduction and other popular applications such as Okular used for reading documents in various formats. To obtain the execution traces from these applications, we first instrumented them along with the framework code of Qt and KDE as discussed in Section 4.1. We then manually exercised their GUIs. In all, the execution time of instrumented applications ranged from 10s to 180s. The collected traces were analyzed for race conditions by SparseRacer.

We report the results on a representative run of each application. Table 2 gives some statistics about the execution traces. The traces contained 1K to 83K operations, with as many as 80K memory accesses. Across these runs, the number of event handlers executed ranged between 15 and 273. Each of the traces contained programmatic event loops. SparseRacer identified the blocks in each of the traces. The number of blocks identified for these traces were orders of magnitude smaller than the number of operations, ranging from 24 to 290. This makes a strong case for reasoning at the granularity of blocks using the sparse representation.

To study the effectiveness of the sparse representation, we built a baseline implementation to compute the complete happens-before relation without using our sparse representation optimization. Both SparseRacer and the baseline implementation use a graph based data-structure to compute the happens-before relation. A previous work [25] proposed an optimization to reduce the number of nodes by coalescing contiguous memory operations without any synchronization operation in between. We adapted this optimization to work with programmatic event loops, and used it in the baseline implementation. We then applied all the rules described in Section 3 until saturation. Pairs inferred by R13 and R14 were represented explicitly in the graph computed by the baseline implementation.

We ran the experiments on an Intel Xeon E5-1620 3.60 GHz machine with 8 cores and 24GB RAM. The baseline implementation and SparseRacer run on a single core.

5.2 Results

5.2.1 Results of Race Detection

For each of the traces in Table 2, SparseRacer reported the interfering handlers which contained pairs of racing operations. In our case, these are pairs of use and free operations on the same memory location. The number of interfering handler pairs reported by SparseRacer varies from 1 to 14 across the applications. In total, SparseRacer reported 58 pairs of interfering handlers for the nine applications.

The procedure to try to exercise each of the races is as follows. We identify the stack contents at the use and free operations and also look up the sequences of events leading up to them. Both these pieces of information are recorded by our instrumentation module as outlined in Section 4.1. We then instrument a build of the application using AddressSanitizer [39], a memory safety tool. We run the instrumented application in the GDB debugger and exercise different scheduling choices for events. If AddressSanitizer detects a use-after-free, we know that the interfering handlers can cause a crash.

Using the strategy outlined, we found that 13 interfering handler pairs are indeed true positives. That is, they can be scheduled in such a way that a use-after-free gets triggered. As shown in Table 2, these included two use-after-free cases for Ark, Cervisia, KOrganizer and Okular. The remaining five applications each had one use-after-free. In all these cases, the use operation was part of an event handler X that spun a programmatic event loop and the handler Y which freed the memory referenced by the use operation could be scheduled in the same event loop. Either the event for Y was not enabled before X could start running or the object was allocated inside X but before the programmatic event loop was started. Thus, the existing race detection techniques which reason only about the default event loop could not have detected these scenarios. The memory safety tools such as Valgrind [29] and AddressSanitizer [39] do not reason about scheduling non-determinism and hence, could not find them given the original traces.

We reported all of the 13 use-after-frees to the developers of the respective applications. The links to the bug reports we filed and subsequent followup are available online [9]. So far, developers have confirmed eight of these to be valid bugs. The developers of KTnef, Kate and KWrite have also fixed the bugs we reported, including the bug discussed in the Introduction. The others are under review at present. Note that we already have evidence that they cause use-after-frees under specific event sequences. Use-after-frees cause application crashes leading to loss of application state. Besides, they are an exploitable form of security vulnerabilities [14].

Our experiments thus show that race conditions can arise due to unintended interference in programmatic event loops and developers care about the resulting bugs. SparseRacer

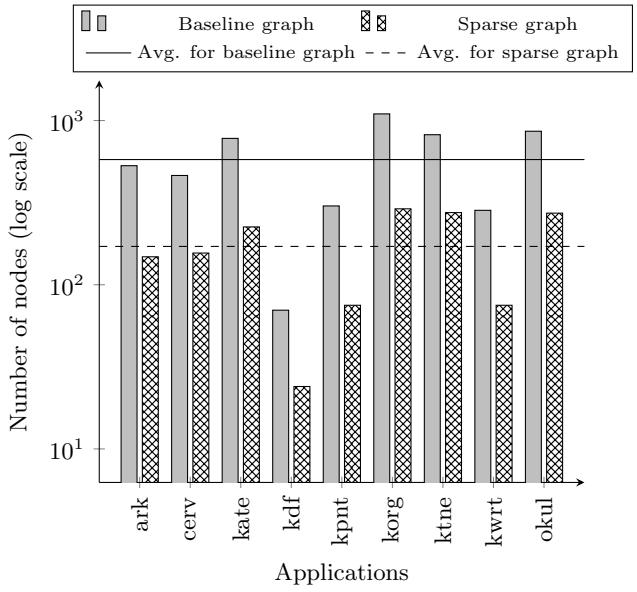

Figure 8: Comparison of the number of nodes.

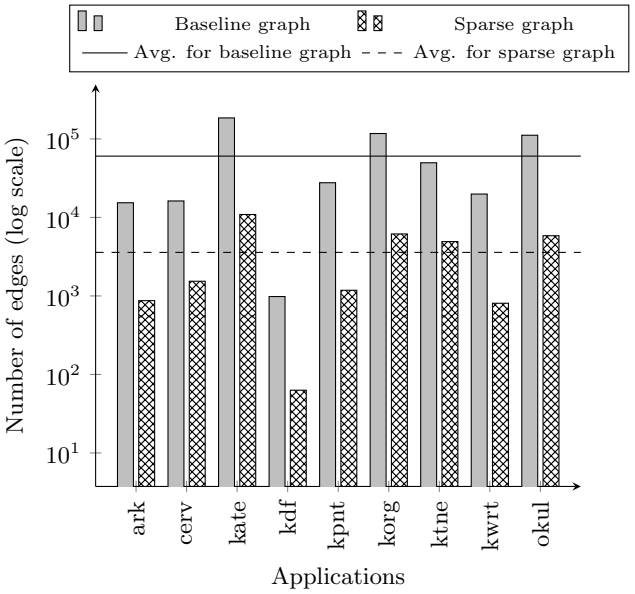

Figure 9: Comparison of the number of edges.

is a practical tool which we could successfully apply on Qt-based KDE applications to find such bugs.

5.2.2 *Performance*

Due to the instrumentation code inserted to collect traces, the applications experienced a minimum slowdown of 1.09X and a maximum slowdown of 2.59X. The average slowdown over all applications was 1.31X.

SparseRacer builds a directed graph of the sparse representation where the nodes are blocks and edges indicate happens-before ordering between blocks. The baseline implementation computes the complete happens-before relation on the graph obtained after the node coalescing optimization described in Section 5.1. We call them respectively as the *sparse graph* and the *baseline graph*. We now compare SparseRacer and the baseline implementation in terms of the number of nodes and edges in their happens-before graphs. Figure 8 plots the number of nodes in these graphs for each of the application traces from Table 2. The number of nodes in the sparse graph is fewer by 70.4% on average compared to the number of nodes in the baseline graph. Overall, it is fewer by a minimum of 65.7% and maximum of 75.2%. Figure 9 shows the number of edges added in the sparse graph compared to the baseline graph after they saturate. There is a drastic reduction in the number of edges ranging from 90.1%–95.9% with an average of 94.1%. This shows that the sparse representation is more compact than the baseline representation. As discussed in Section 3, any happens-before ordering can be inferred from the sparse representation in constant time. Thus, this reduction does not incur precision loss or penalty in terms of looking up the happens-before ordering between arbitrary operations.

This translates into reduced analysis time if the sparse representation is used. The time taken by SparseRacer on each of the runs is given in Table 2. SparseRacer took 2 minutes on an average, and a maximum of 8 minutes. Figure 10 compares the time taken to compute the happens-before graphs by SparseRacer and the baseline implementation. SparseRacer offers an average speedup of 5x over the baseline implementation, with a minimum speedup of 2x and a maximum speedup of 48x. This presents strong evidence that the sparse representation of the happens-before relation improves efficiency of race detection.

5.3 Limitations

SparseRacer currently reasons only about the main thread of an application. We plan to extend it to multi-threaded programs in the future. For the present work, the main limitation is the presence of false positives. While the number of false positives is not overwhelmingly large, for some applications such as Cervisia and Okular, SparseRacer reported 14 pairs of handlers containing racing operations but only 2 were found to be true positives. Overall, there were 45 false positives and 13 true positives across the nine applications. The light-weight pruning tactics outlined in Section 4.2 helped reduce benign cases of use-free races.

A majority of false positives arose because of cases where environmental events had causal relations, that we did not model. For example, the event closing a dialog can occur only after the event that displays it. We plan to model these constraints to eliminate false positives due to them. To avoid use-after-frees in C++ code, it is a common practice to reference count objects. An object is not freed unless its reference count is zero. Due to the associated performance overheads, usually only a small set of objects are reference-counted. SparseRacer does not make a distinction between the ordinary objects and reference-counted objects. In a few cases, SparseRacer had flagged use-free races on reference-counted objects. Another potential source of false positives is ad-hoc synchronization. A handler that frees a memory location may set a guard variable that is tested before use of that location in other handlers. Recent work [37, 19] has proposed techniques to deal with such cases. Though we did not observe cases of this form, we think it will be useful to implement those techniques in SparseRacer.

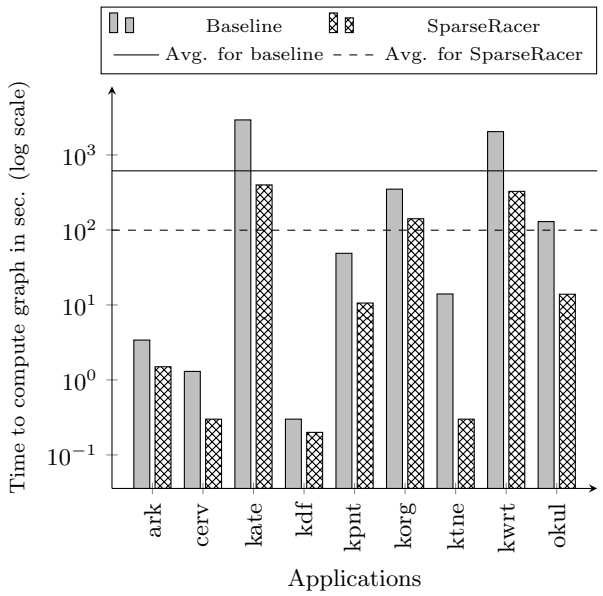

Figure 10: Comparison of the time taken to compute the happens-before graphs.

6. RELATED WORK

Multi-threaded programs. There is a large body of work on race detection for multi-threaded programs using 1) locksets [38, 17, 28, 36], 2) happens-before reasoning [20, 18], and 3) hybrid approaches combining the two [30, 34, 43, 42, 41]. These approaches do not capture the event-loop based scheduling semantics of event-driven programs. Simulating events through fresh threads either does not scale since the number of events in a usual run is large or produces many false positives [37]. The presence of programmatic event loops is likely to compound these issues further.

Event-driven programs. There is growing interest in race detection for event-driven programs. The existing approaches consider only the case of default event loops. This paper addresses the combined scheduling scheme of the default and programmatic event loops. We are not aware of any prior work that considers this scheduling mechanism.

WebRacer [33] formalizes the event-driven semantics of JavaScript programs and EventRacer [37] introduces an efficient race detection algorithm for this model. Zheng et al. [47] describe a static analysis to detect races in JavaScript applications arising out of the special case of asynchronous AJAX calls. These approaches do not model programmatic event loops because JavaScript programs have only the default event loop. Similarly, recent techniques for race detection for Android applications [19, 25, 15] can reason about multi-threaded programs but with only default event loops.

We cannot simply treat each block in a trace as a separate event in order to obtain traces involving only the default event loop. As an example, consider the trace from Figure 6. The fragments of `e1` and `e2` after the respective resumes are denoted by blocks B7 and B6 respectively. We can treat them as new events, say $\texttt{e1}'$ and $\texttt{e2}'$, and enqueue them from the respective pause operations (operations 8 and 10). We then end up enqueuing $\texttt{e1}'$ before $\texttt{e2}'$, whereas, B7 executes *after* B6. Postponing the enqueues of $\texttt{e1}'$ and $\texttt{e2}'$ to resets also runs into the same issue since the reset for `e1` (operation 12) occurs *before* that of `e2` (operation 15). Thus, the technique presented in this work, which directly reasons about programmatic event loops is essential.

Kahlon et al. [22] propose a static analysis to detect races in multi-threaded C programs with event-based communication. Their approach presents a points-to analysis to identify the handlers passed around through function pointers.

Concurrency-related memory errors. Memory errors are considered harmful as they can cause crashes and can be exploited to breach software security. The effect-oriented approach [46] and the work on detection of use-free races in Android applications [19] therefore target their efforts towards concurrency-related memory errors. Our work follows the same line and targets use-free races but under a different scheduling scheme.

GUI testing and analysis. Many researchers have proposed techniques to automate GUI testing and analysis [16, 44], including those aimed at detecting performance problems [35] and invalid thread accesses [45]. While we obtained traces by manually exercising the GUI, it would be interesting to investigate whether systematic GUI testing approaches can be targeted at uncovering race conditions, amplifying the effect of predictive race detectors such as SparseRacer.

7. CONCLUSIONS AND FUTURE WORK

This paper analyzes the scheduling mechanism of programmatic event loops. This mechanism is used extensively in popular software frameworks such as web browsers, GUI libraries and some OS APIs. Programmatic event loops introduce complex interleavings of events that are absent when only the runtime can dequeue events in its default event loop. We have presented a set of happens-before rules to reason about reorderings of operations in the presence of programmatic event loops. We have also designed a sparse representation of the happens-before relation to improve efficiency of race detection. We presented a practical race detector called SparseRacer and used it to find several real bugs in open-source applications arising from unintended interference in programmatic event loops.

There are many future directions. It would be interesting to extend our technique to also reason about multiple threads. While it is certainly possible to derive suitable happens-before rules, how to best exploit the sparse representation would be a challenging problem. Event-driven programming is a very rich space and we want to investigate principled extensions to the programmatic event loop semantics by careful study of real-world software.

8. ACKNOWLEDGMENTS

This work is partially supported by a faculty award from Mozilla Corporation. The first author was supported by an IBM PhD Fellowship.

9. REFERENCES

[1] http://doc.qt.io/qt-4.8/qeventloop.html.
[2] docs.wxwidgets.org/3.0/classwx_event_loop_base.html.

[3] http://docs.oracle.com/javase/7/docs/api/java/awt/SecondaryLoop.html.
[4] https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsIThread.
[5] https://www.chromium.org/developers/design-documents/threading.
[6] https://msdn.microsoft.com/en-us/library/system.windows.forms.application.doevents(v=vs.110).aspx.
[7] developer.apple.com/library/mac/documentation/corefoundation/Reference/CFRunLoopRef.
[8] http://sourceforge.net/projects/ktnef/.
[9] http://www.iisc-seal.net/qt-bug-reports.
[10] https://okular.kde.org/.
[11] http://www.unix.org/version4/iso_std.html.
[12] http://www.qt.io/.
[13] https://www.kde.org.
[14] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit. *Black Hat USA*, 2007.
[15] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for android applications. In *OOPSLA*, pages 332–348. ACM, 2015.
[16] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640. ACM, 2013.
[17] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232. ACM, 2000.
[18] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133. ACM, 2009.
[19] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *PLDI*, pages 326–336. ACM, 2014.
[20] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, Nov. 1999.
[21] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, Dec. 1990.
[22] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *FSE*, pages 13–22. ACM, 2009.
[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, 1978.
[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. IEEE Computer Society, 2004.
[25] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *PLDI*, pages 316–325. ACM, 2014.
[26] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.
[27] Z. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura. *Programming Android*. O'Reilly, 2012.
[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319. ACM, 2006.
[29] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.
[30] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, pages 167–178. ACM, 2003.
[31] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at USENIX, 1996.
[32] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX*, pages 199–212. USENIX Association, 1999.
[33] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, pages 251–262. ACM, 2012.
[34] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP*, pages 179–190. ACM, 2003.
[35] M. Pradel, P. Schuh, G. C. Necula, and K. Sen. EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*, pages 33–47, 2014.
[36] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *TOPLAS*, 33(1):3:1–3:55, 2011.
[37] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166. ACM, 2013.
[38] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, pages 391–411, 1997.
[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. USENIX ATC, pages 309–318. USENIX Association, 2012.
[40] S. Souders. *Even faster web sites: performance best practices for web developers*. O'Reilly Media, Inc., 2009.
[41] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, pages 369–384. ACM, 2011.
[42] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *FSE*, pages 205–214. ACM, 2007.
[43] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234. ACM, 2005.
[44] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE*, pages 396–405. IEEE Computer Society, 2007.
[45] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ISSTA*, pages 243–253, 2012.
[46] W. Zhang, C. Sun, J. Lim, S. Lu, and T. W. Reps. ConMem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. *ACM Trans. Softw. Eng. Methodol.*, 22(2):10, 2013.
[47] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *WWW*, pages 805–814. ACM, 2011.