Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers

Pantazis Deligiannis
Department of Computing,
Imperial College London, UK
p.deligiannis@imperial.ac.uk

Alastair F. Donaldson
Department of Computing,
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

Zvonimir Rakamarić
School of Computing,
University of Utah, USA
zvonimir@cs.utah.edu

***Abstract*—Concurrency errors, such as data races, make device drivers notoriously hard to develop and debug without automated tool support. We present WHOOP, a new fully automatic approach that statically analyzes drivers for data races. WHOOP is empowered by *symbolic pairwise lockset analysis*, a novel analysis that can soundly detect all potential races in a driver. Our analysis avoids reasoning about thread interleavings and thus can scale well. Exploiting the race-freedom guarantees provided by WHOOP, we achieve a sound partial-order reduction that can significantly accelerate CORRAL, an industrial-strength bug-finder for concurrent programs. Using the combination of WHOOP and CORRAL, we analyzed 16 drivers from the Linux 4.0 kernel, showing that we can achieve 1.5–20× speedups over standalone CORRAL.**

I. INTRODUCTION

Device drivers are complex pieces of system-level software, operating at the thin boundary between hardware and software to provide an interface between the operating system and hardware devices that are attached to a computer. Drivers are notoriously hard to develop and debug [1]. This has a negative impact on hardware product releases, as time-to-market is commonly dominated by driver development, verification and validation schedules [2]. Even after a driver has shipped, it typically has many undetected errors: Chou et al. [3] gathered data from 7 years of Linux kernel releases and found that the relative error-rate in driver source code is up to 10 times higher than in any other part of the kernel, while Swift et al. [4] found that 85% of the system crashes in Windows XP are due to faulty drivers. Regarding *concurrency bugs*, a recent study [5] found that they account for 19% of the total bugs in Linux drivers, showcasing their significance. The majority of these concurrency bugs were found to be *data races* or *deadlocks* in various configuration functions and hot-plugging handlers.

Concurrency bugs are exacerbated by the complex and hostile environment in which drivers operate [1]. The OS can invoke drivers from multiple threads, a hardware device can issue interrupt requests that cause running processes to block and switch execution context, and the user may remove a device (hot-plugging) while some operation is still running. These scenarios can cause *data races* if insufficient synchronization mechanisms are in place to control concurrent access to shared resources. Data races are a source of undefined behavior in C [6, p. 38], and lead to nondeterministically occurring bugs that can be hard to reproduce, isolate and fix, especially in the context of complex operating systems. Several techniques have been successfully used to analyze device drivers [7], [8], [9], [10], [11], [12], [13], [14], but most focus on generic sequential program properties and protocol bugs. Linux kernel analyzers, such as sparse [15], coccinelle [16] and lockdep [17], can find deadlocks in kernel source code, but are unable to detect races. Techniques that can detect races in drivers [18], [19], [20], [21], [14] are usually either *unsound* (i.e. can miss real bugs) or *imprecise* (i.e. can report false bugs), and typically sacrifice precision for scalability. Thus, there is a clear need for new tools that are able to detect races efficiently and precisely.

We present WHOOP, an automated approach for static data race analysis in device drivers. WHOOP is empowered by *symbolic pairwise lockset analysis*, which attempts to prove a driver race-free by: (i) deriving a sound *sequential* program that *over-approximates* the originally concurrent program; (ii) instrumenting it to record *locksets*; and (iii) using the locksets to assert that all accesses to the same shared resource are consistently protected by a common lock. Reducing analysis to reasoning over a sequential program avoids the need to enumerate thread interleavings, and allows reuse of existing successful sequential verification techniques. We show that we can exploit the race-freedom guarantees provided by our symbolic analysis to achieve a sound partial-order reduction that significantly accelerates CORRAL [14], a precise bug-finder used by Microsoft to analyze drivers and other concurrent programs. Using WHOOP and CORRAL we analyzed 16 drivers from the Linux 4.0 kernel. By combining WHOOP and CORRAL we achieve analysis speedups in the range of 1.5–10× for most of our benchmarks, compared with using CORRAL in isolation. For two drivers, we observed even greater speedups of 12× and 20×. WHOOP currently supports Linux drivers, but our approach is conceptually generic and could be applied to analyze drivers for other operating systems, as well as concurrent systems that use a similar programming model (e.g. file systems).

To summarize, our contributions are as follows:

- We propose symbolic pairwise lockset analysis, a sound and scalable technique for automatically verifying the absence of data races in device drivers.
- We present WHOOP, a tool that leverages our approach for analyzing data races in device drivers.
- We show that we can achieve a sound partial-order reduction using our technique to accelerate CORRAL, an industrial-strength bug-finder.
- We analyze 16 Linux drivers, showing that WHOOP is efficient at race-checking and accelerating CORRAL.

```
static loff_t nvram_llseek(struct file *file,
    loff_t offset, int origin) {
  switch (origin) {
    case 0: break;
    case 1: offset += file->f_pos; break; // racy
    default: offset = -1;
  }
  if (offset < 0) return -EINVAL;
  file->f_pos = offset; // racy
  return file->f_pos; // racy
}
```

Fig. 1. Racy entry point in the generic_nvram Linux driver

II. Background

Concurrency in Device Drivers Modern operating systems address the demand for responsiveness and performance in device drivers by providing multiple sources of concurrency [1]: an arbitrary number of entry points from the same driver can be invoked concurrently; a running driver process can block, causing the driver to switch execution to another process; and hardware interrupts can be handled concurrently. These forms of concurrent execution are prone to *data races*.

Definition 1: A *data race* occurs when two distinct threads access a shared memory location, at least one of the accesses modifies the location, at least one of the accesses is non-atomic, and there is no mechanism in place to prevent these accesses from being simultaneous.

Figure 1 shows a racy entry point, `nvram_llseek`, in the generic_nvram Linux driver. The driver can invoke the entry point concurrently from two threads T_1 and T_2, with the same `file` struct as a parameter. However, this can lead to multiple possible data races, because T_1 and T_2 can access the `f_pos` field of `file` in arbitrary order. Our tool, WHOOP (see §III), was able to find these races automatically (see §V).

The most common method for avoiding races is by protecting sets of program statements that access a shared resource with *locks*, forming *critical sections*. Figure 2 shows how to use locking to eliminate the races in Figure 1. Because the `return` statement can potentially race on the `f_pos` field, we store the result in a temporary variable `res` inside the critical section. Careless use of locks has many well-known pitfalls [22]: coarse-grained locking can hurt performance, as it limits the opportunity for concurrency; while fine-grained locking can easily lead to deadlocks. Although the Linux kernel provides sophisticated lock-free synchronization mechanisms [1, p. 123], such as atomic read-modify-write operations, in this work we focus on locks as they are widely used.[1]

Lockset Analysis Lockset analysis is a lightweight race detection method proposed in the context of Eraser [23], a dynamic data race detector. The idea is to track the set of locks that are *consistently* used to protect a memory location during program execution. If that lockset ever becomes empty, the analysis reports a *potential* race on that memory location. This is because an empty lockset suggests that a memory location *may* be accessed simultaneously by two or more threads.

Lockset analysis for a concurrent program starts by creating a *current* lockset CLS_T for each thread T of the program, and a lockset LS_s for each shared variable s used in the program. Initially, CLS_T is empty for every thread T, because the threads do not hold any locks on program start. The lockset LS_s for each variable s is initialized to the set of all locks manipulated by the program, because initially each access to s has (vacuously) been protected by every lock. The program is executed as usual (with threads scheduled according to the OS scheduler), except that instrumentation is added to update locksets as follows. After each *lock* and *unlock* operation by T, CLS_T is updated to reflect the locks currently held by T. When T accesses shared variable s, any locks that are not common to LS_s and CLS_T are removed from LS_s. If LS_s becomes empty as a result, a warning is issued that the access to s may be unprotected.

[1]We treat lock-free operations soundly, by not regarding them as providing any protection between threads, but this can lead to false alarm race reports.

```
static loff_t nvram_llseek(struct file *file,
    loff_t offset, int origin) {
  mutex_lock(&nvram_mutex); // lock
  switch (origin) {
    case 0: break;
    case 1: offset += file->f_pos; break;
    default: offset = -1;
  }
  if (offset < 0) {
    mutex_unlock(&nvram_mutex); // unlock
    return -EINVAL;
  }
  file->f_pos = offset;
  loff_t res = file->f_pos; // store result
  mutex_unlock(&nvram_mutex); // unlock
  return res;
}
```

Fig. 2. Using a lock to eliminate the data races in the previous example

Figure 3 shows an example of applying lockset analysis to a concurrent program consisting of two threads T_1 and T_2, both accessing a global variable A. Initially, LS_A, which is the lockset for A, contains all possible locks in the program: M and N. During execution of T_1, the thread writes A without holding N, and thus N is removed from LS_A. Next, during execution of T_2, the thread writes A without holding M, and thus LS_A becomes empty. As a result, a warning is reported because the two threads do not consistently protect A.

In contrast to more sophisticated race analyses that encode a *happens-before* relation between threads [24] (e.g. using vector clocks), lockset analysis is easy to implement, lightweight, and thus has the potential to scale well. The technique, though, suffers from imprecision (i.e. can report false bugs), because a violation of the locking discipline does not always correspond to a real data race [23], [25], [26], [27], [28]. Furthermore, the code coverage in dynamic lockset analyzers, such as Eraser, is limited by the execution paths that are explored under a given scheduler. This is because the tool runs the program in a truly concurrent environment, thus it only gets to see what the thread did on the particular schedule that was followed.

To counter the above limitations, techniques such as Locksmith [20] and RELAY [21] have explored the idea of applying lockset analysis in a static context, using dataflow analysis. In this paper, we present a novel symbolic lockset analysis method that involves generating verification conditions, which are then discharged to a theorem prover.

III. The WHOOP Approach

We now present *symbolic pairwise lockset analysis*, a novel method for data race analysis in device drivers. In §III-A we

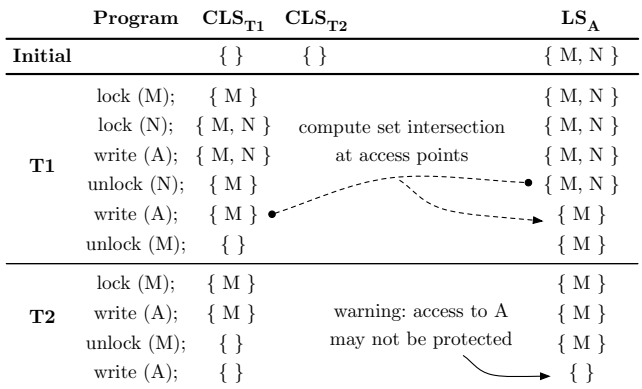

Fig. 3. Applying lockset analysis on a concurrent program

describe how the approach works in a semi-formal manner, with respect to a simple concurrent programming model. In §III-B we explain how we have implemented our analysis in a practical tool, WHOOP, that can be applied directly to driver source code. Our experimental evaluation (§V) demonstrates that WHOOP has value as a stand-alone analyzer, and that results from our analysis can be exploited to significantly boost the performance of a more precise symbolic analysis for concurrency, offered by the CORRAL [14] tool; we discuss the latter approach in §IV.

A. Symbolic Pairwise Lockset Analysis

Our approach considers, for a given driver, every pair of entry points that can potentially execute concurrently. For each such pair, we use symbolic verification to check whether it is possible for the pair to race on a shared memory location. We soundly model the effects of additional entry points by treating the driver shared state abstractly: when an entry point reads from the shared state, a nondeterministic value is returned. Restricting to pairs of entry points, rather than analyzing all entry points simultaneously, exploits the fact that data races occur between pairs of threads and limits the complexity of the generated verification conditions.[2] The trade-off is that a quadratic number of entry point pairs must be checked. In §III-B we discuss optimizations based on device driver domain knowledge to reduce the number of pairs to some extent.

Symbolic verification of a pair of entry points works by (i) instrumenting each entry point with additional state to record locksets, and (ii) attempting to verify a sequential program that executes the instrumented entry points in sequence and then asserts, for each shared location, that the locksets for each entry point with respect to this location have a non-empty intersection. Verification of the resulting sequential program can be undertaken using any sound method; in practice we employ the Boogie verification engine [29], which requires procedure specifications and loop invariants to be generated, after which verification conditions [30] (VCs) are generated and discharged to an automated theorem prover.

We now detail the approach in a semi-formal manner, in the context of a simple concurrent programming model.

[2]In principle, our approach could be applied at a coarser level of granularity: e.g. by considering all entry points one after the other, taking into account that an entry point can race with itself. However, using pairwise analysis has the additional advantage that it enables us to easily run the analysis for each pair in parallel (for performance), although we leave this for future work.

Statement	Notes
$x = e;$	private assignment, where $x \in V_T$ and e is an expression over V_T
$x = f(\overline{e});$	procedure call, where $x \in V_T$, $\overline{e}$ is a sequence of expressions over V_T, f is the name of a procedure in $procs_T$
$s = e;$	shared write, where $s \in V_s$ and e is an expression over V_T
$x = s;$	shared read, where $x \in V_T$ and $s \in V_s$
$\mathsf{lock}(m);$	mutex lock, where $m \in M$
$\mathsf{unlock}(m);$	mutex unlock, where $m \in M$

Fig. 4. The allowed statements in our simple programming model

Concurrent Programming Model We consider a concurrent programming model where an unbounded number of threads execute a set of pre-defined *thread templates*. At any given point of execution a certain number of threads are active, each thread executing a particular template. In the context of device drivers, a thread template corresponds to a driver entry point, and multiple instances of the same thread template may execute concurrently, just as multiple invocations of a single driver entry point may be concurrent. Further threads may start executing at any point during execution; in the context of device drivers this corresponds to the OS invoking additional driver entry points.[3] For ease of presentation only, our model does not feature aggregate data types, pointers or dynamic memory allocation. These *are* handled by our implementation, and in §III-B we discuss interesting practical issues arising from the handling of a full-blown language.

A *concurrent program* is described by a finite set of *shared* variables V_s, a finite set of mutexes M, and a finite set of *thread templates*. A thread template T consists of a finite set of procedures $procs_T$ and a finite set of private variables V_T. A designated procedure $main_T \in procs_T$ denotes the starting point for execution of T by a thread. Each procedure of $procs_T$ is represented by a control flow graph of basic blocks, where each block contains a sequence of statements. A basic block either has a single successor or a pair of successors. In the latter case, an *exit condition* over thread-private variables determines the successor to which control should flow on block exit.

The allowed statements are shown in Figure 4, and include designated statements for reading from and writing to shared variables. In particular, shared variables may not appear in arbitrary expressions. This restriction simplifies our presentation of lockset instrumentation below, and a program that does not satisfy this restriction can be trivially pre-processed into one that does via the introduction of additional private temporary variables to record values read from the shared state. We do not specify the form of expressions, nor the types of variables, assuming a standard set of data types and operations.

Semantics Let $\mathcal{I}$ be an infinite set from which dynamic thread ids will be drawn. The state of a running concurrent program consists of: a valuation of the shared variables V_s; a mapping that associates each mutex in M with an id from $\mathcal{I}$, recording which thread currently holds the mutex, or with a special value $\perp \notin \mathcal{I}$ to indicate that the mutex is not held by any thread; and a list of *threads*. Each thread consists of an id, drawn from $\mathcal{I}$,

[3]We do *not* consider the case where one thread spawns another thread, which does not occur in the context of drivers; rather we aim to capture the scenario where additional threads are launched by the environment.

Original Statement	Instrumented Statement
$s = e;$	$W_i = W_i \cup \{s\};$ $LS_{s,i} = LS_{s,i} \cap CLS_i;$
$x = s;$	$R_i = R_i \cup \{s\};$ $LS_{s,i} = LS_{s,i} \cap CLS_i;$ $\mathsf{havoc}(x_i);$
$\mathsf{lock}(m);$	$CLS_i = CLS_i \cup \{m\};$
$\mathsf{unlock}(m);$	$CLS_i = CLS_i \setminus \{m\};$

Fig. 5. Instrumenting statements for lockset analysis

a thread template T an index indicating the next statement of T to be executed by the thread, and a valuation of the thread private variables, V_T. If multiple threads are instances of the same template T, then each thread carries a *separate* valuation of the private variables for this template.

Initially, the valuation of shared variables is arbitrary, no mutexes are held (i.e. each mutex maps to $\perp$), and the list of threads is empty. At any point of execution, a new thread may be added to the list of threads. This involves selecting a thread template T and an id $i \in \mathcal{I}$ that has not been previously used during program execution, setting the point of execution for the new thread to be the first statement of $main_T$, and choosing an arbitrary valuation for the private variables V_T. We consider a standard interleaving model of concurrency: at any execution point, a thread may execute its current statement, unless that statement has the form $\mathsf{lock}(m)$ and mutex m is held by some thread. Executing a statement causes the state of the thread, and the shared state, to be updated in a standard manner. For example, if a thread following template T executes $s = e$, where $s \in V_s$ and e is an expression over V_T, the shared variable valuation is updated so that s has the value determined by evaluating e in the context of the thread's private variable valuation. Because our interest is in data race analysis for race-free programming, we are not concerned with relaxed memory behavior: race-free programs exhibit only sequentially consistent behaviors.

A thread terminates if it reaches the end of $main_T$, and is removed from the list of threads. Since our interest is in analysis of device drivers, which are reactive, we do not consider the notion of global program termination.

Lockset Instrumentation For templates T and U (including the possibility that T and U are equal) we want to check whether it is possible for a thread executing T to race with a thread executing U, in the presence of arbitrarily many further concurrently executing threads. To this end, we first *instrument* a template T for lockset analysis (see §II). Given an arbitrary symbol i, we define the instrumentation of T with respect to i, denoted T_i. There are two aspects to this instrumentation phase: *renaming* and *lockset instrumentation*. Renaming is straightforward: all occurrences of each private variable $x \in V_T$ used in T are replaced with a renamed variable x_i in T_i, and every procedure $f \in procs_T$ is renamed (both at its declaration site and at all call sites) to f_i in T_i. The purpose of renaming is to ensure that when we analyze a pair of templates, T and U, both templates execute distinct procedures and operate on distinct private data. This is vital in the case where T and U are the same.

Lockset instrumentation introduces: sets $R_i \subseteq \mathcal{P}(V_s)$ and $W_i \subseteq \mathcal{P}(V_s)$ to track the shared variables that have been read from and written to, respectively, by the thread executing T; a current lockset $CLS_i \subseteq \mathcal{P}(M)$ to record the mutexes currently held by the thread; and, for each shared variable $s \in V_s$, a lockset $LS_{s,i}$ to record the mutexes that are consistently held when the thread accesses s. The statements of each procedure in T_i that access shared variables and mutexes are instrumented to manipulate these sets, as shown in Figure 5. For a shared variable assignment $s = e$, we record in W_i that s has been written to, and update $LS_{s,i}$ to eliminate any mutexes that are not currently held (those mutexes that are not in CLS_i). A shared variable read $x = s$ is instrumented analogously, with an additional havoc command which we discuss below. Instrumentation of mutex manipulation commands, $\mathsf{lock}(m)$ and $\mathsf{unlock}(m)$, involves updating CLS_i to add and remove mutex m, respectively.

$$
\begin{array}{l}
CLS_i = \emptyset;\ R_i = \emptyset;\ W_i = \emptyset; \\
CLS_j = \emptyset;\ R_j = \emptyset;\ W_j = \emptyset; \\
\texttt{for } s \in V_s \texttt{ do } LS_{s,i} = M;\ LS_{s,j} = M; \\
main_{T_i}(); \\
main_{U_j}(); \\
\texttt{assert } \forall s \in V_s\ . \\
\quad s \in W_i \cap (R_j \cup W_j) \vee s \in W_j \cap (R_i \cup W_i) \implies \\
\quad\quad LS_{s,i} \cap LS_{s,j} \neq \emptyset;
\end{array}
$$

Fig. 6. The sequential program to be analyzed in order to prove race-freedom for a pair of thread templates

Shared State Abstraction Recall that while our aim is to perform race analysis for pairs of threads, we must be sure to account for possible side-effects due to other threads that are running concurrently. The instrumentation of Figure 5 achieves this via *nondeterminism*: when reading from a shared variable s, a nondeterministic value is returned. This is reflected by the use of a havoc command, which sets its argument to an arbitrary value. Because all shared state accesses are abstracted in this fashion, it is possible to completely dispense with the shared variables after the lockset instrumentation has been performed. As a result, when instrumenting a shared variable write, the effect of the write is not explicitly modeled.

Sequentialization The pseudocode of Figure 6 shows the sequential program that we analyze in order to prove race-freedom for a pair of thread templates T and U. Assuming that T and U have been instrumented using distinct symbols i and j, yielding T_i and U_j, the sequential program operates as follows. First, the read, write and current locksets for T_i and U_j are initialized to be empty, and for each shared variable s, the locksets $LS_{s,i}$ and $LS_{s,j}$ are initialized to the full set of mutexes, M. The main procedures of the instrumented thread templates, $main_{T_i}$ and $main_{U_j}$, are then executed in turn (the order does not matter, due to renaming). Finally, an assertion checks for consistent use of mutexes: if s is written during execution of T_i and accessed during execution of U_j, or vice-versa, then the locksets $LS_{s,i}$ and $LS_{s,j}$ must contain at least one common mutex.

Soundness We sketch an argument that if the program of Figure 6 is correct (i.e. the assertion, which is — described above — at the end of the program holds), then it is impossible for a thread executing template T to race with a thread executing template U, under the assumption that the threads are guaranteed to terminate. Let us assume that the program of Figure 6 is correct, and suppose (by way of contradiction) that a thread executing T can in fact race with a thread executing U,

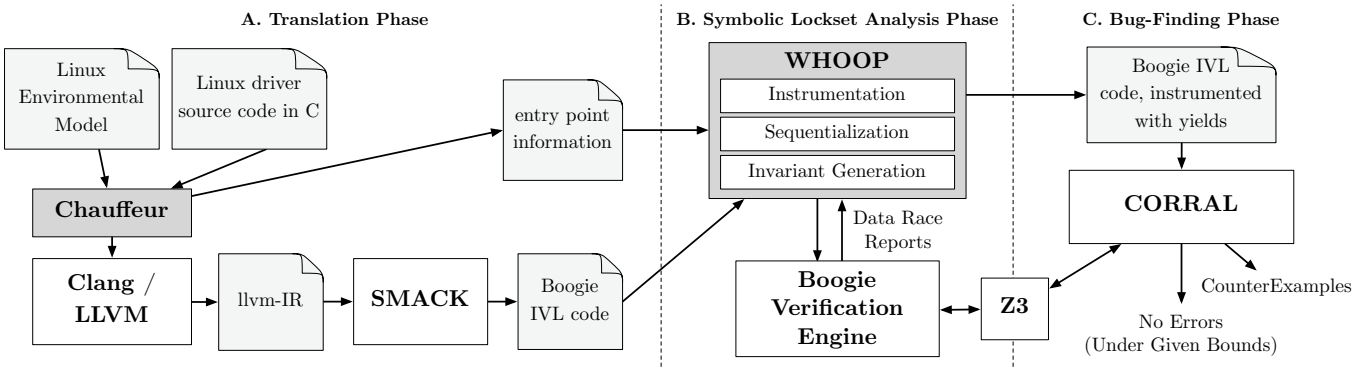

Fig. 7. The WHOOP architecture, empowered by state-of-the-art compilation (Clang/LLVM and SMACK) and verification (Boogie and CORRAL) tools

on some shared variable s. By our hypothesis that the program is correct, and that the threads terminate, the assertion checked at the end of the program guarantees that at least one mutex, say m, belongs to both $LS_{s,i}$ and $LS_{s,j}$. By the definition of a lockset (and according to the manner in which shared accesses are instrumented in Figure 5), this means that m is held during every access to s by both T_i and U_j. As a result, m must be unlocked and locked between the two accesses, which contradicts that the pair of accesses is racing.

In the presence of non-termination the assertion at the end of Figure 6 may not be reached. The termination analysis problem for device drivers has been widely studied (see e.g. [11]), and in the remainder of the paper we do not consider termination issues, assuming that the drivers we analyze in our experimental evaluation (see §V) are terminating.

B. Implementation in WHOOP

The simple concurrent programming model of §III-A is deliberately idealistic to make it easy to describe our symbolic verification technique. In practice, Linux drivers are written in C, our technique does not know up-front which are the driver entry points, drivers do not work with a cleanly specified set of named locks, and rather than having a given set of named shared variables, we have arbitrary memory accesses via pointers. We now explain how we have taken the conceptual ideas from §III-A and used them to build WHOOP[4], a practical, fully automatic tool for detecting data races in drivers.

Architecture Figure 7 depicts the WHOOP toolchain. The input to WHOOP is a Linux driver written in C, together with an environmental model[5] that is required to "close" the driver so that it can be analyzed for races. Initially, WHOOP uses three LLVM[6]-based tools, Chauffeur[7], Clang[8] and SMACK[9] [31], to translate the driver and its environmental model into an abstract program written in the Boogie intermediate verification language (IVL) [32], a simple imperative language with well-defined semantics that is used as the input to a number of cutting-edge verifiers (e.g. Boogie and CORRAL). Next, WHOOP instruments and sequentializes the program to perform symbolic pairwise lockset analysis (see Figure 7 – B and §III-A) using the Boogie verification engine. After the verification phase ends, WHOOP can exploit any inferred race-freedom guarantees to accelerate precise race-checking with CORRAL (see Figure 7 – C and §IV).

```
const struct file_operations nvram_fops = {
  .llseek = nvram_llseek,
  .read = read_nvram,
  .write = write_nvram,
  .unlocked_ioctl = nvram_unlocked_ioctl
};
```

Fig. 8. Entry point definitions in the generic_nvram driver

We engineered the Chauffeur and WHOOP components of our toolchain (denoted with grey boxes in Figure 7). For the remaining components, we were able to reuse industrial-strength tools that are robust and battle-proven via their use in many complex software projects.

Extracting Entry Point Information Chauffeur is a Clang-based tool that traverses the abstract syntax tree (AST) of the original driver and extracts all entry point identifier names, together with the identifier names of their corresponding kernel API functions. Linux drivers define entry points in a standard way (see Figure 8 for an example of how the generic_nvram driver defines the entry points for the `file_operations` API); Chauffeur identifies these definitions in the AST and outputs the relevant information in an XML file, which is parsed by WHOOP to be used during the instrumentation.

Translation for Verification Next, the driver is compiled by Clang to LLVM-IR [33], a low-level assembly-like language in single static assignment (SSA) form. Function calls (e.g. for locking and unlocking) are preserved in this translation and, hence, we do not need to keep track of them separately. SMACK then translates the driver from LLVM-IR to Boogie IVL, which is the input language of WHOOP. An important feature of SMACK is that it leverages the pointer-alias analyses of LLVM to efficiently model the heap manipulation operations of C programs in Boogie IVL. Thus, WHOOP does not need to directly deal with pointers and alias analysis, a hard problem on its own right, which allows us to reuse robust existing techniques and focus instead on verification efforts.

To achieve scalability, SMACK uses a *split* memory model that exploits an alias analysis to soundly partition memory locations into non-overlapping equivalence classes that do not alias. This has been shown to lead to more tractable verification compared with a *monolithic* model where the heap is consid-

[4]https://github.com/pdeligia/whoop
[5]This consists of stub header files modeling relevant Linux kernel APIs.
[6]http://llvm.org
[7]https://github.com/mc-imperial/chauffeur
[8]http://clang.llvm.org
[9]https://github.com/smackers/smack

ered to be an array of bytes [34]. The split memory model is based on *memory regions*, which are maps of integers that model the heap. A benefit of using this model is that distinct memory regions denote disjoint sections of the heap. We leverage this knowledge inside WHOOP to guide and optimize our lockset instrumentation and analysis, and to create a fine-grained context-switch instrumentation as discussed in §IV.

Identifying Locks When the instrumentation phase begins, WHOOP performs an inter-procedural static analysis (at Boogie IVL level) to identify all available locks and rewrite each one (both at declaration and at all access sites) to a unique constant Boogie variable. The reason behind this transformation is that representing all locks statically, instead of their original SMACK pointer-based representation, helps WHOOP perform various internal instrumentations and optimizations. Currently, WHOOP only supports mutexes and spinlocks that are available in the Linux kernel APIs. However, it is relatively easy to enhance our tool with knowledge of other locking primitives. If WHOOP cannot infer a lock, e.g. because it was created dynamically or was indexed from an array of locks, it will exit with a warning. There are two reasons behind this: (i) it is arguably hard to detect such locks via static analysis; and (ii) a small, fixed number of locks is advocated by Linux experts as good practice when developing drivers [1, p. 123].

Watchdog Race-Checking Instrumentation Data race detection is performed by introducing sets containing the locks that are consistently held when accessing each shared variable and sets containing all shared variables that are read and written (see §III-A and the instrumentation of Figure 5). These sets can be modeled directly in Boogie as characteristic functions, using maps. However, this requires the use of quantifiers to express properties related to set contents. For instance, to express that a set X of elements of type A is empty, where X is represented as a map from A to Bool, we would require the quantified expression $\forall a : A : \neg X[a]$. It is well known that automated theorem proving in the presence of quantified constraints is challenging, and that theorem provers such as Z3 [35] are often much more effective when quantifiers are avoided.

To avoid quantifiers and the associated theorem proving burden, we use instead a *watchdog race-checking instrumentation*, adapted from [36]. Suppose we are analyzing entry points T and U, and that after translation into Boogie these entry points share a common memory region, MR. When analyzing T and U for races, we introduce an unconstrained symbolic constant $watched_{MR}$, representing some unspecified index into MR; we call this the *watched offset* for MR. We then attempt to prove that it is impossible for T and U to race on MR at index $watched_{MR}$. If we can succeed in proving this, we know that T and U must be race-free for the *whole* of MR, since the watched offset was arbitrary. This technique of choosing an arbitrary index to analyze for each map manipulated by an entry point pair can be seen as a form of quantifier elimination: rather than asking the underlying theorem prover to reason for all indices of MR, in a quantified manner, we eliminate the quantifier in our encoding, and instead ask the theorem prover to reason about a single, but arbitrary, index of MR.

Generating Loop and Procedure Summaries Early in the development of WHOOP, we experimented with analyzing recursion-free drivers using full inlining. We found that this did not scale to large drivers: when we tried to apply WHOOP to the r8169 ethernet driver (see §V), we quickly exhausted the memory limits of our experimental platform as the driver exhibited recursion, which cannot be fully inlined.

To make our analysis scale while maintaining precision, and to support recursion, we use the Houdini algorithm [37] to automatically compute summaries (pre- and post-conditions and loop invariants). Given a pool of *candidate* pre-conditions, post-conditions, and loop invariants, Houdini repeatedly attempts to verify each procedure. Initially, the entire candidate pool is considered. If verification fails due to an incorrect candidate, this candidate is discarded. The process repeats until a fixpoint is reached. The (possibly empty) set of remaining candidates has been proven to hold, and can be used to summarize calls and loops during further program analysis.

Houdini does not generate the initial pool of candidates: WHOOP generates them using a set of heuristics, and passes them to Houdini as a starting point. The idea is to automatically generate likely program invariants based on syntactic patterns extracted from an inter-procedural pass over the code for an entry point. We give two examples; for clarity we use notation from the simple shared variable concurrent programming model of §III-A. If we observe syntactically that procedure f of entry point T may write to, but does not read from, shared variable s, then when instrumenting T with symbol i, we guess $s \in W_i$ and $s \notin R_i$ as post-conditions for f_i. These guesses may be incorrect, for instance if the potential write to s turns out to be in dead code, or if a read from s has already been issued on entry to f_i. Similarly, if syntactic analysis indicates that f may unlock mutex m, we guess $m \notin CLS_i$ as a post-condition for f_i; this guess may be wrong, for instance if the unlock operation is not reachable or if a subsequent lock operation acquires the mutex again. We stress that guessing incorrect candidate invariants does not compromise the soundness of verification: WHOOP is free to speculate candidates that are later deemed to be incorrect, and thus discarded by Houdini. The balance we try to strike is to have WHOOP generate sufficient candidates to enable precise lockset analysis, without generating so many candidates that the speed of the Houdini algorithm is prohibitively slow.

In practice, the candidates are generated at the Boogie IVL level, with respect to SMACK-generated memory regions and referencing the auxiliary variables introduced by our watchdog race-checking instrumentation.

Verification and Error Reporting For each entry point pair, the instrumented sequential program, equipped with procedure and loop summaries, is sent to the Boogie verification engine. For each procedure in the program, Boogie generates a VC and discharges it to the Z3 theorem prover. In particular, the verification for the root-level procedure, encoding the sequential program sketched in Figure 6, encodes the race-freedom check for the entry point pair. Successful verification implies that the pair is race-free, while an error (i.e. counterexample) denotes a *potential* data race and is reported to the user. To improve usability, WHOOP has a built-in error reporter that matches counterexamples to source code. The following is a race that WHOOP found and reported for the example of Figure 1:

```
generic_nvram.c: error: potential read-write race:
  read by entry point nvram_llseek, generic_nvram.c:54:2
    return file->f_pos;
  write by entry point nvram_llseek, generic_nvram.c:53:2
    file->f_pos = offset;
```

Optimizations We have implemented various optimizations to increase the precision and performance of WHOOP. We comment on the two most effective optimizations.

First, we enriched WHOOP with information regarding *kernel-imposed serialization*, to increase precision. The Linux kernel can serialize calls to entry points, thus forcing them to run in sequence instead of an interleaved manner. For example, a large number of networking entry points are mutually serialized with RTNL, a network-specific kernel lock. We discovered this when WHOOP reported many races between a number of networking entry points of the r8169 driver (see §V); when we investigated the source of these races, we found out that these entry points could not race in reality because of RTNL. WHOOP exploits this knowledge and does not create pairs for entry points that are mutually serialized by the kernel. This is an ongoing manual effort: the more drivers we study, the more such properties we discover to make WHOOP more precise. Currently, this information is hard-coded inside WHOOP. In the future, we plan to provide a configuration file where the user can denote such kernel-specific information.

Second, we soundly reduce the number of memory regions that are analyzed for races. If memory region MR is accessed by only one entry point in a pair then, trivially, the pair cannot race on MR. We thus disable lockset analysis for MR. This can reduce the complexity of VCs that need to be solved by the theorem prover, speeding up the verification process.

Practical Assumptions Related to Soundness WHOOP is "soundy"[10] [38]: it aims in principle to perform a sound analysis that can prove absence of races, but suffers from some known sources of unsoundness, which we now comment on.

We assume that the formal parameters of an entry point do not alias, and thus cannot race. This is a potentially unsound feature that can be turned off using a command line option. Without this assumption we have observed WHOOP reporting false alarms, and in our experience so far we have not missed any races by assuming non-overlapping parameters. We also rely on the soundness of our best-effort environmental model, and on exploiting domain-specific knowledge related to entry point serialization by the Linux kernel.

As well as inheriting soundness issues arising from currently unknown bugs in WHOOP and in the external tools that WHOOP relies on, we acknowledge that: (i) SMACK is subject to sources of unsoundness, e.g. it models integers as an infinite set (rather than as a finite set of bit-vectors), and its memory model can potentially be unsound in (typically rare) situations where programs use unaligned byte-level memory accesses; and (ii) that the combination of Clang and SMACK commits our approach to specific choices related to undefined and implementation-defined aspects of the C language when translating to Boogie. However, WHOOP makes no fundamental assumptions related to these translation decisions, meaning that a more accurate C-to-Boogie translation would automatically lead to a more accurate analysis with WHOOP.

Limitations As a lockset analyzer, WHOOP can be imprecise because a violation of the locking discipline does not always correspond to a real race (e.g. when lock-free synchronization is used). WHOOP also uses over-approximation, which can be another source of imprecision. Furthermore, the tool does not check for dynamically created locks or for locks provided by external libraries, although the later could be addressed by providing a mechanism for users to declare custom locks. We also do not currently treat interrupt handlers in a special way; we just assume that they can execute concurrently with any entry point. One way to address this is to model interrupt-specific kernel functions (e.g. for enabling/disabling interrupts).

Another limitation of WHOOP is that it is unable to verify drivers designed to be accessed by only a single process at a time. This *single-open device* [1, p. 173] mode can be enforced by atomically testing (at the beginning of an entry point) a global flag that indicates device availability: if the flag is set to true, then the checking entry point executes, else it blocks. Because WHOOP over-approximates read accesses to shared variables, and thus this flag, it can falsely report a pair as racy. However, [1, p. 173] advises against serializing drivers in this way, as it "inhibits user ingenuity" (e.g. a user might expect that a driver can be accessed concurrently for performance).

Statically analyzing drivers requires "closing" the environment by abstracting away the low-level implementation details. To this end, we developed a simple model for the Linux kernel that consists of (nondeterministic) stub functions. A limitation of our model is that it can, and will, ultimately result in false positives. However, we currently only focus on finding data races, and thus can get away with over-approximating large parts of the underlying kernel functionality, without losing too much precision. Making our model more precise is an ongoing manual effort, but requires Linux expertise. We argue that further work on the model is orthogonal to the contributions of this paper. Also, even if our symbolic analysis results in false positives, WHOOP can still use the results to significantly speedup a more precise bug-finder, as seen in §IV and §V.

IV. ACCELERATING PRECISE RACE-CHECKING

WHOOP is a sound but imprecise static race analyzer. For developers who deem false alarms as unacceptable, we consider a method for leveraging the full or partial race-freedom guarantees provided by WHOOP to accelerate CORRAL [14], a precise bug-finder used by Microsoft to analyze Windows drivers [39]. Because CORRAL operates on Boogie programs, it was easy to integrate it into our toolchain (see Figure 7 – C). Our technique, though, is general and capable in principle of accelerating any concurrency bug-finder that operates by interleaving threads at shared memory operations.

CORRAL is a symbolic bounded verifier for Boogie IVL that uses the Z3 SMT solver to statically reason about program behaviors. It checks for violations of provided assertions, and reports a precise counterexample if an assertion violation is found. CORRAL performs bounded exploration of a concurrent program in two steps. First, given a bound on the number of allowed context-switches, the concurrent program is appropriately *sequentialized*, and the generated sequential version preserves reachable states of the original concurrent program [40], [41], [42]. Then, CORRAL attempts to prove bounded (in terms of the number of loop iterations and recursion depth) sequential reachability of a bug in a goal-directed, lazy fashion to postpone state space explosion when analyzing a large program. It uses two key techniques to achieve this: (i)

[10] http://soundiness.org/

variable abstraction, where it attempts to identify a minimal set of shared variables that have to be precisely tracked in order to discharge all assertions; and (ii) stratified inlining, where it attempts to inline procedures on-demand as they are required for proving program assertions.

Race-Checking Instrumentation To detect data races with CORRAL, WHOOP outputs a Boogie program instrumented with a simple, but effective encoding of race checks. Whenever there is a write access to a shared variable s, WHOOP instruments the program as follows:

```
s = e;          // original write
yield;          // allow for a context-switch
assert s == e; // check written value
```

Likewise, whenever there is a read access to s, WHOOP instruments the program as follows:

```
x = s;          // original read
yield;          // allow for a context-switch
assert x == s; // check read value
```

A `yield` statement denotes a nondeterministic context-switch, and is used by CORRAL to guide the sequentialization.

CORRAL is inherently unsound (i.e. can miss real races), because it performs bounded verification. However, CORRAL is precise and, assuming a precise environmental model, it will only report true races. WHOOP takes advantage of this precision to report only feasible data races. Also note that our instrumentation conveniently tolerates most benign races, as it does not report a write-write race if two write accesses update the same shared memory location with the same value.

In this work, we use CORRAL to analyze individual pairs of entry points. We do not use any abstraction to model additional threads, as we want CORRAL to report only true races. Because we only analyze pairs, though, CORRAL will miss races that require more than two threads to manifest. We could extend our setup so that more than two threads are considered by CORRAL, but because the number of threads that an OS kernel might launch is unknown in general, we are inevitably limited by some fixed maximum thread count.

Sound Partial-Order Reduction By default, and assuming no race-freedom guarantees, WHOOP instruments a `yield` after each shared memory access of each entry point, and after every lock and unlock operation.[11] WHOOP then sends this instrumented program to CORRAL, which leverages sequentialization to explore all possible thread interleavings up to a pre-defined bound. Our approach to accelerating CORRAL is simple and yet effective: if, thanks to WHOOP's analysis, we know that a given statement that accesses shared memory cannot be involved in a data race, then we do not instrument a `yield` after this statement. This is a form of *partial order reduction* [43], and reduces the number of context-switches that CORRAL must consider in a *sound* manner: there is no impact on the bugs that will be detected. This is because each access, for which a `yield` is suppressed, is guaranteed to be protected by some lock (a consequence of lockset analysis). If the access is a write, its effects are not visible by the other entry point in the pair until the lock is released. If the access is a read, the value of the shared location cannot change until the lock is released. The fact that a `yield` is placed after each unlock operation suffices to take account of communication between entry points via the shared memory location.

[11] We acknowledge that in the presence of data races and relaxed memory, even considering all interleavings of shared memory accesses may be insufficient to find all bugs.

TABLE I. PROGRAM STATISTICS AND RACE-CHECKING RESULTS FROM APPLYING WHOOP AND CORRAL ON OUR BENCHMARKS.

				WHOOP		CORRAL
Benchmarks	**LoC**	**#Pairs**	**#MRs**	**#Racy Pairs**	**#Racy MRs**	**#Races Found**
generic_nvram	283	14	39	7	2	4
pc8736x_gpio	354	27	55	13	6	5
machzwd	457	10	49	6	3	1
ssu100	568	7	27	✗	✗	✗
intel_scu_wd	632	10	45	5	1	2
ds1286	635	15	49	5	3	✗
dtlk	750	21	53	10	6	✗
fs3270	883	15	54	9	1	✗
gdrom	890	94	41	21	2	✗
swim	996	28	80	15	7	8
intel_nfcsim	1272	10	24	10	2	✗
ps3vram	1499	4	32	1	1	✗
sonypi	1729	30	62	19	4	2
sx8	1751	2	47	2	1	1
8139too	2694	46	37	40	4	✗
r8169	7205	192	50	88	1	✗

We have implemented two different `yield` instrumentations in WHOOP: Yield-EPP and Yield-MR. The first instruments `yield` statements in a binary fashion: if WHOOP proves an entry point pair as race free, then it will instrument a `yield` only after each lock and unlock statement of the pair; else if WHOOP finds that a pair might race, then it will instrument a `yield` after all visible operations of the pair. Yield-MR is a finer-grained instrumentation: it instruments a `yield` only after each access to a memory region that might race in the pair (regardless if the pair has not been fully proven as race-free), and after each lock and unlock. In our experiments (see §V), Yield-MR significantly outperforms Yield-EPP.

Our partial-order reduction can be used to accelerate CORRAL for arbitrary bug-finding in concurrent programs. In the remainder of this paper, though, we only regard races as bugs and use CORRAL solely to look for races.

V. EVALUATION

We performed experiments to validate the usefulness of the WHOOP approach (§III) and its combination with CORRAL (§IV). We first present race-checking results from running WHOOP and CORRAL on 16 drivers taken from the Linux 4.0 kernel.[12] We then evaluate the runtime performance and scalability of CORRAL using different yield instrumentations and context-switch bounds. Our results demonstrate that WHOOP can efficiently accelerate race-checking with CORRAL.

For reproducibility, we make all of our tools (including source) and benchmarks available online:

http://multicore.doc.ic.ac.uk/tools/Whoop/ASE2015

Experimental Setup We performed all experiments on a 3.40GHz Intel Core i7-2600 CPU with 16GB RAM running Ubuntu Linux 12.04.5 LTS, LLVM 3.5, SMACK 1.5,

[12] https://www.kernel.org

TABLE II. COMPARISON WITH DIFFERENT YIELD INSTRUMENTATION GRANULARITIES AND CONTEXT-SWITCH BOUNDS.

	WHOOP	CORRAL				WHOOP + CORRAL							
		Yield-ALL — Time (sec)				Yield-EPP — Time (sec)				Yield-MR — Time (sec)			
Benchmarks	**Time (sec)**	**#Y**	**csb = 2**	**csb = 5**	**csb = 9**	**#Y**	**csb = 2**	**csb = 5**	**csb = 9**	**#Y**	**csb = 2**	**csb = 5**	**csb = 9**
generic_nvram	2.3	92	32.0	67.7	197.5	47	17.7	24.2	132.1	29	13.5	16.9	49.3
pc8736x_gpio	4.1	691	169.3	595.1	27337.6	500	79.4	432.4	22514.1	167	41.3	79.4	1358.9
machzwd	2.8	104	38.1	45.6	78.4	51	26.1	31.2	55.2	22	15.7	17.3	23.9
ssu100	2.9	82	11.1	13.8	37.3	✗	3.1	3.1	3.2	✗	3.1	3.1	3.1
intel_scu_wd	2.4	314	22.8	130.3	1571.7	217	14.7	70.0	748.0	196	13.5	66.1	689.3
ds1286	4.1	513	35.0	40.2	51.8	245	22.3	25.7	33.0	129	14.5	16.1	19.3
dtlk	5.4	801	182.6	263.7	793.0	664	91.2	150.7	633.2	286	41.6	52.9	104.6
fs3270	3.2	321	81.5	419.4	T.O.	239	62.6	405.7	33468.4	211	40.7	295.3	8883.0
gdrom	9.5	2058	388.0	390.8	392.1	1143	104.0	105.5	107.1	812	99.2	102.4	107.6
swim	5.8	1270	271.0	2746.6	T.O.	996	164.9	2309.8	T.O.	805	95.5	1847.9	T.O.
intel_nfcsim	3.6	732	39.8	85.0	1539.9	732	44.4	89.6	1543.4	601	21.0	32.0	278.0
ps3vram	4.5	189	99.6	2376.8	T.O.	176	96.2	2249.0	T.O.	156	55.8	1698.6	T.O.
sonypi	10.6	1745	1966.5	T.O.	T.O.	1566	1924.4	T.O.	T.O.	1024	906.0	T.O.	T.O.
sx8	2.8	227	12.9	15.2	21.4	227	16.1	18.5	24.5	217	15.9	18.5	24.3
8139too	18.9	7151	548.2	14469.0	T.O.	6266	474.7	12677.3	T.O.	4964	359.8	5664.5	T.O.
r8169	144.5	16035	T.O.	T.O.	T.O.	11723	T.O.	T.O.	T.O.	10535	T.O.	T.O.	T.O.

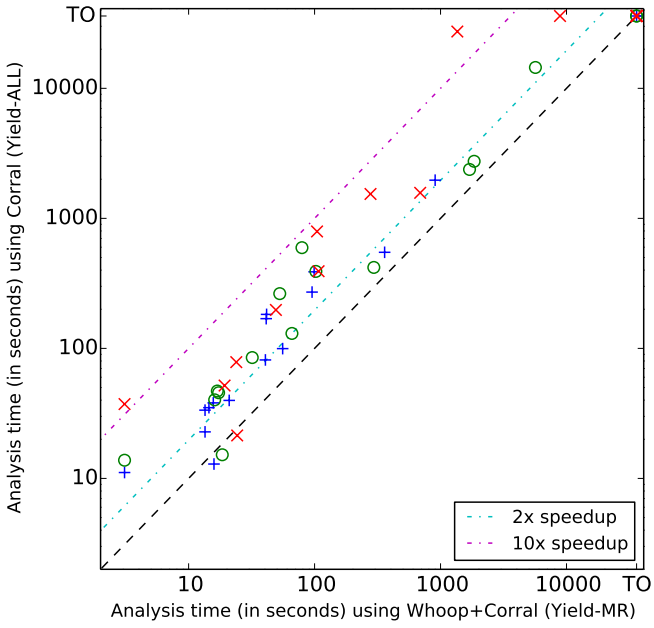

Fig. 9. Scatter plot showing the runtime speedups that CORRAL achieves using WHOOP with the Yield-MR instrumentation. The symbols +, ∘ and ×, represent a context-switch bound of 2, 5 and 9, respectively.

Z3 4.3.2, Boogie rev. 4192 and CORRAL rev. 534. We also used Mono 4.1.0 to run Boogie and CORRAL.

Benchmarks We evaluate our methodology against 16 drivers from the Linux 4.0 kernel. We chose non-trivial drivers of several types: block; char; ethernet; near field communication (nfc); universal serial bus (usb); and watchdog. For all these drivers, we had to understand their environment and manually model it; this required approximately two months of work.

Race-Checking Table I presents statistics for all our benchmarks: lines of code (LoC); number of entry point pairs (#Pairs); number of SMACK memory regions (#MRs); number of racy pairs (#Racy Pairs) and number of racy memory regions (#Racy MRs) reported by WHOOP; and number of races discovered by CORRAL using a context-switch bound (csb) of 2 (#Races Found). Using a higher csb did not uncover any further races; this might mean that races in our benchmarks require only one or two context-switches to manifest, or that CORRAL hit its bounds before discovering a deeper bug. CORRAL did not discover any races that WHOOP did not already report. This is expected, as WHOOP aims for soundness, and increases our confidence in the implementation of WHOOP.

We can see that WHOOP reports more races than CORRAL does. This is because WHOOP employs an over-approximating shared state abstraction to conservatively model the effects of additional threads when analyzing an entry point pair, and because lockset analysis is inherently imprecise; both factors can lead to false positives. On the other hand, CORRAL is precise, but can miss races because only a limited number of context-switches are considered. Another issue with CORRAL is loop coverage due to unsound loop unrolling. To tackle this, we enable the built-in loop over-approximation described in [39]. This can potentially lead CORRAL to report false bugs, but we have not seen this in practice. Finally, when we apply CORRAL to a pair of entry points, we just check the specific pair and do not account for the effects of other threads (see §IV); this can also cause CORRAL to miss some races.

Most of the races that WHOOP and CORRAL discovered fall into two categories. The first is about accessing a global counter (or flag) from concurrent entry points, without holding a lock. This might be for performance, and indeed a lot of the races we found might be benign. Even benign races, though, lead to undefined behavior according to the C standard, and it is well known that undefined behaviors can lead to unexpected results when combined with aggressive compiler optimizations. The second is about an entry point modifying a field of a struct (either global or passed as a parameter) without holding a lock. This can lead to a race if another entry point simultaneously accesses the same field of the same struct.

As an example of the second category, we found the following race in the generic_nvram driver (see Figure 1): the `llseek` entry point accesses the file offset `file->f_pos` without a lock (`file` is passed as a parameter to `llseek`).

This causes a race if the driver invokes `llseek` from another thread passing the same `file` object as a parameter. We observed that another char driver, using the same APIs, *does* use a mutex to protect the offset access in its `llseek` entry point, leading us to suspect that the race we found in generic_nvram is not benign. We have filed a bug report in the Linux kernel bug tracker and are awaiting a response.

Accelerating CORRAL Table II presents runtime results from using WHOOP, CORRAL and WHOOP + CORRAL to analyze our benchmarks, while Figure 9 plots the runtime speedups that CORRAL achieves using WHOOP with the Yield-MR instrumentation. All reported times are in seconds and averaged over three runs. CORRAL was configured with a time budget of 10 hours (T.O. denotes timeout) and a csb of 2, 5 and 9. Standalone CORRAL uses Yield-ALL, which instruments context-switches (i.e. `yield` statements) after all visible operations, while WHOOP + CORRAL uses Yield-EPP and Yield-MR, which instrument context-switches in a more fine-grained fashion (see §IV). The table also shows the number of context-switches per instrumentation (#Y).

WHOOP uses over-approximation to achieve scalability and, as expected, executes significantly faster than CORRAL in all our benchmarks. For example, CORRAL times out in all configurations (with and without WHOOP) when trying to analyze the r8169 ethernet driver, while WHOOP manages to analyze the driver in 144.5 seconds. We believe that the reason behind this is that r8169 has deeply-nested recursion in some of its entry points, which might get CORRAL stuck. This is not an issue for WHOOP, which uses procedure summarization. This shows that WHOOP has value as a stand-alone analyzer.

Using the race-freedom guarantees from WHOOP, we managed to significantly accelerate CORRAL in most of our benchmarks; the best results were achieved using Yield-MR. Figure 9 shows that most speedups using Yield-MR are between 1.5× and 10×; while in ssu100, and pc8736x_gpio respectively, we achieved a speedup of 12×, and 20× respectively, using a csb of 9. We noticed that a higher csb typically results into greater speedups when exploiting WHOOP. This is expected, as a higher csb directly translates into a larger sequentialized program, and thus WHOOP has more opportunities to reduce this sequentialization. However, if WHOOP fails to verify any race-related properties of a driver, it might slow down CORRAL. We noticed this in the sx8 driver: WHOOP verified 46 out of 47 memory regions, but it did not fully verify any of the pairs; CORRAL, on the other hand, analyzed the only two pairs of the driver quite quickly (21.4 seconds with csb of 9). We believe that in this case the overhead of running WHOOP outweighed the benefits of using Yield-MR.

Other Tools We tried to compare WHOOP with other similar approaches (see §VI). However, we found this to be hard in practice: we downloaded Locksmith [20], but could not get it to work with the 4.0 Linux drivers (the tool was last updated in 2007 and might be missing some of the latest features); we also could not find source code or binaries for [44], [45], [46].

VI. RELATED WORK

Static race analysis is a promising alternative to dynamic techniques, which restrict analysis to the schedule chosen by a (possibly controlled) scheduler, providing limited coverage [47]. Warlock [48] and LockLint [49] are notable static race analyzers. In comparison to WHOOP, these tools rely heavily on user annotations. Most related to our lockset analysis are the static lockset analyzers RELAY [21] and Locksmith [20]. Both tools, though, have several limitations. RELAY found 5022 warnings when analyzing the Linux kernel, with only 25 of them being true data races. To tackle this issue, RELAY employs unsound post-analysis filters and, hence, can also filter out real bugs. Locksmith was successfully applied in several small Pthreads applications and 7 medium-sized Linux device drivers, but the authors reported that the tool was unable to run on several large programs, showcasing its limited scalability. WHOOP aims to achieve scalability *and* precision: the first, via novel symbolic pairwise lockset analysis; and the second, by accelerating CORRAL, an industrial-strength precise bug-finder.

Prior works on static race checking also include: [50], which combines static analysis and runtime access caching to speedup dynamic race detection; [44], which uses a divide-and-conquer algorithm that partitions all pointers of a program that do not alias in disjoint sets to achieve a scalable race analysis; [45], which uses abstract interpretation to achieve a sound partial-order reduction on the set of thread interleavings and statically reduce the number of false race warnings; and [46], which employs inter-procedural alias analysis and verifiable user annotations to split programs into disjoint sections, based on non-communicating accesses of shared data, and eliminate redundant checks during dynamic race detection. Our tool, WHOOP, uses symbolic lockset analysis, which involves generating verification conditions and discharging them to a theorem prover, and then uses CORRAL to filter out false races.

Our pairwise approach to analysing driver entry points, employing abstraction to model additional threads, was inspired by the *two thread reduction* used by the GPUVerify tool in the analysis of data-parallel OpenCL and CUDA kernels [51], [36]. The idea of pairwise analysis of components in a concurrent system has been broadly applied, notably in model checking of cache coherence protocols [52].

VII. CONCLUSIONS

In this paper we presented WHOOP, a new fully automatic approach for detecting all possible data races in device drivers. Compared to traditional data race detection techniques that are based on happens-before and typically attempt to explore as many thread interleavings as possible (and thus face code coverage and scalability issues), WHOOP uses over-approximation and symbolic pairwise lockset analysis, which has the potential to scale well. Exploiting the race-freedom guarantees provided by WHOOP, we showed that we can achieve a sound partial-order reduction that can significantly accelerate CORRAL, a state-of-the-art concurrent bug-finder. Combining WHOOP and CORRAL, we analyzed 16 drivers from the Linux 4.0 kernel, showing that our technique can analyze drivers 1.5–20× faster than standalone CORRAL.

REFERENCES

[1] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers (Third Edition)*. O'Reilly, 2005.

[2] R. Yavatkar, "Era of SoCs," Presentation at the Intel Workshop on Device Driver Reliability, Modelling and Synthesis, 2012.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 73–88.

[4] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 207–222.

[5] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: Taming device drivers," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 275–288.

[6] ISO/IEC, "Programming languages - C," International Standard, 2011.

[7] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the 2006 EuroSys Conference*, 2006, pp. 73–85.

[8] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI-C programs using SAT," *Formal Methods in System Design*, vol. 25, no. 2-3, pp. 105–127, 2004.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, 2000.

[10] T. A. Henzinger, G. C. Necula, R. Jhala, G. Sutre, R. Majumdar, and W. Weimer, "Temporal-safety proofs for systems code," in *Proceedings of the 14th International Conference on Computer Aided Verification*, 2002, pp. 526–538.

[11] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 2006, pp. 415–426.

[12] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010.

[13] 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*, 2012.

[14] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Proceedings of the 24th International Conference on Computer Aided Verification*, 2012, pp. 427–443.

[15] J. Corbet, "Finding kernel problems automatically," https://lwn.net/Articles/87538/, 2004.

[16] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008, pp. 247–260.

[17] J. Corbet, "The kernel lock validator," https://lwn.net/Articles/185666/, 2006.

[18] D. R. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 237–252.

[19] S. Qadeer and D. Wu, "KISS: Keep it simple and sequential," in *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004, pp. 14–24.

[20] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Context-sensitive correlation analysis for race detection," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 320–331.

[21] 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*, 2007, pp. 205–214.

[22] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[25] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," in *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003, pp. 179–190.

[26] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003, pp. 167–178.

[27] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware Java runtime," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 245–255.

[28] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 121–133.

[29] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Proceedings of the 4th International Symposium on Formal methods for Components and Objects*, 2006, pp. 364–387.

[30] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, 2005, pp. 82–87.

[31] Z. Rakamarić and M. Emmi, "SMACK: Decoupling source language details from verifier implementations," in *Proceedings of the 26th International Conference on Computer Aided Verification*, 2014, pp. 106–113.

[32] R. DeLine and K. R. M. Leino, "BoogiePL: A typed procedural language for checking object-oriented programs," Microsoft Research, Tech. Rep., 2005.

[33] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[34] Z. Rakamarić and A. J. Hu, "A scalable memory model for low-level code," in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2009, pp. 290–304.

[35] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[36] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer, "Engineering a static verification tool for GPU kernels," in *Proceedings of the 26th International Conference on Computer Aided Verification*, 2014, pp. 226–242.

[37] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Proceedings of the International Symposium of Formal Methods for Increasing Software Productivity*, 2001, pp. 500–517.

[38] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[39] A. Lal and S. Qadeer, "Powering the Static Driver Verifier using Corral," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 202–212.

[40] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-Bounded Scheduling," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011, pp. 411–422.

[41] S. K. Lahiri, S. Qadeer, and Z. Rakamarić, "Static and precise detection of concurrency errors in systems code using SMT solvers," in *Proceedings of the 21st International Conference on Computer Aided Verification*, vol. 5643, 2009, pp. 509–524.

[42] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," in *Proceedings of the 20th International Conference on Computer Aided Verification*, 2008, pp. 37–51.

[43] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032.

[44] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007, pp. 226–239.

[45] V. Kahlon, S. Sankaranarayanan, and A. Gupta, "Semantic reduction of thread interleavings in concurrent programs," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 124–138.

[46] M. Das, G. Southern, and J. Renau, "Section based program analysis to reduce overhead of detecting unsynchronized thread communication," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 283–284.

[47] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 267–280.

[48] N. Sterling, "WARLOCK - A static data race analysis tool," in *Proceedings of the Usenix Winter 1993 Technical Conference*, 1993, pp. 97–106.

[49] Oracle Corporation, "Analyzing program performance with Sun WorkShop, Chapter 5: Lock analysis tool," http://docs.oracle.com/cd/E19059-01/wrkshp50/805-4947/6j4m8jrnd/index.html, 2010.

[50] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 258–269.

[51] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: a verifier for GPU kernels," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012, pp. 113–132.

[52] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1999, pp. 219–237.