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 W\textit{HOOOP}, a new automated approach that statically analyzes drivers for data races. W\textit{HOOOP} is empowered by \textit{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 scales well. Exploiting the race-freedom guarantees provided by W\textit{HOOOP}, we achieve a sound partial-order reduction that significantly accelerates CORRAL, an industrial-strength bug-finder for concurrent programs. Using the combination of W\textit{HOOOP} and CORRAL, we analyzed 16 drivers from the Linux 4.0 kernel, achieving 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 the 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] established that they account for 19% of the total bugs in Linux drivers, showcasing their significance. The majority of these concurrency bugs were 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 overly 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 can detect races efficiently and precisely.

We present W\textit{HOOOP}, an automated approach for static data race analysis in device drivers. W\textit{HOOOP} is empowered by \textit{symbolic pairwise lockset analysis}, which attempts to prove a driver race-free by: (i) deriving a sound \textit{sequential} program that \textit{over-approximates} the originally concurrent program; (ii) instrumenting it to record \textit{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 W\textit{HOOOP} and CORRAL we analyzed 16 drivers from the Linux 4.0 kernel. Our combination achieves analysis speedups in the range of $1.5–10 \times$ for most of our benchmarks, compared with using CORRAL in isolation. For two drivers, we observed even greater speedups of $12 \times$ and $20 \times$. W\textit{HOOOP} 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 W\textit{HOOOP}, 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 W\textit{HOOOP} is efficient at race-checking and accelerating CORRAL.
Concurrency in Device Drivers  Modern operating systems address the demand for responsiveness and performance in device drivers by providing multiple sources of concurrency: 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 is non-atomic, and there is no mechanism in place to prevent these accesses from being simultaneous.

Fig. 1 shows a racy entry point, `nvram_llseek`, in the generic_nvram Linux 4.0 device driver. This entry point can be invoked concurrently from two threads, with the same `file` struct as a parameter. This can lead to multiple possible data races because the threads can access the `f_pos` field of `file` in arbitrary order. Our tool, W HOOP (see § III-B), 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. Fig. 2 shows how to use locking to eliminate the races in Fig. 1. 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 (e.g. atomic read-modify-write operations), in this work we focus on locks as they are widely used.

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 (i.e. lockset) that are consistently used to protect a memory location during program execution. An empty lockset suggests that a memory location may be accessed simultaneously by two or more threads. Consequently, the analysis reports a potential race on that memory location.

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. Every `CLS_T` is initially empty because the threads

```
static loff_t nvram_llseek(struct file *file,
   loff_t offset, int origin) {
   switch (origin) {
      ...
   case 1: offset += file->f_pos; break; /* racy */
   ...
   } 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 4.0 driver.

```
static loff_t nvram_llseek(struct file *file,
   loff_t offset, int origin) {
   mutex_lock(&nvram_mutex); // lock
   switch (origin) {
      ...
   case 1: offset += file->f_pos; break; /* racy */
   ...
   } 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. Introducing a lock to eliminate the races in the previous example.

do not hold any locks when they start executing. In addition, every `LS_s` is initialized to the set of all locks manipulated by the program since initially each access to `s` is (vacuously) 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 `s`, `LS_s` is updated to the intersection of `LS_s` with `CLS_T`, which removes any locks that are not common to the two locksets. If `LS_s` becomes empty as a result, a warning is issued that the access to `s` may be unprotected.

In contrast to more sophisticated race analyses that encode a happens-before relation between threads (e.g. using vector clocks), lockset analysis is easy to implement, lightweight, and thus has the potential to scale well. The technique, though, can report false alarms since a violation of the locking discipline does not always correspond to a real data race [24, 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.

To counter the above limitations, techniques such as Locksmith [29] and RELAY [31] have explored the idea of applying lockset analysis statically, 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 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

---

1We treat lock-free operations soundly by regarding them as not providing any protection between threads; this can lead to false alarms.
practical tool, WHOOP, that can be applied directly to driver source code.

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 threads 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. 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. A post-condition for the sequential program asserts, for every 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 (VCs) [30] 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.

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. 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. 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 \( V_\tau \), a finite set of mutexes \( M \), and a finite set of thread templates. A thread template \( T \) consists of a finite set of procedures \( \text{procs}_T \) and a finite set of private variables \( V_T \). A designated procedure \( \text{main}_T \in \text{procs}_T \) denotes the starting point for execution of \( T \) by a thread. Each procedure of \( \text{procs}_T \) 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 Fig. 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 is trivial to enforce by pre-processing. We do not specify the form of expressions, nor the types of variables, assuming a standard set of data types and operations.

Semantics Let \( I \) 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 \( V_\tau \); a mapping that associates each mutex in \( M \) with an id from \( I \), recording which thread currently holds the mutex, or with a special value \( \perp \notin I \) to indicate that the mutex is not held by any thread; and a list of thread states. Each thread state has an associated thread template \( T \), the type of the thread (and multiple threads may have the same associated template), and consists of an id (drawn from \( I \)), 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.

\( ^2 \)In principle, our approach could be applied at a coarser level of granularity, such as 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.

\( ^3 \)We do not consider the case where one thread spawns another thread, which does not typically occur in the context of drivers; rather we aim to capture the scenario where additional threads are launched by the environment.
Initially, the valuation of shared variables is arbitrary, no mutexes are held (i.e., each mutex maps to \( \bot \)), and the list of thread states is empty. At any point of execution, a new thread state may be added to the list of thread states. This involves selecting a thread template \( T \) and an id \( i \in I \) that has not been previously used during program execution, setting mutexes are held (i.e., each mutex maps to the private variables held by the thread; and, for each shared variable \( s \) that access shared variables and mutexes are instrumented to manipulate these sets, as shown in Fig. 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, lock\((m)\) and unlock\((m)\), involves updating \( CLS_i \) to add and remove mutex \( m \), respectively.

### Lockset Instrumentation

For templates \( T \) and \( U \), which can be 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 \( T \) a template \( T \) for lockset analysis (see [2]). 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: each occurrence of a private variable \( x \) in \( V_T \) used in \( T \) is replaced with a renamed variable \( x_i \) in \( T_i \), and every procedure \( f \) in \( \text{procs}_T \) 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 \( T_i \) that access shared variables and mutexes are instrumented to manipulate these sets, as shown in Fig. 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, lock\((m)\) and unlock\((m)\), involves updating \( CLS_i \) to add and remove mutex \( m \), respectively.

### Soundness

We sketch an argument that if the program of Fig. 6 is correct (i.e., the assertion, described above, holds invariantly at the end of the program), 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 Fig. 6 is correct, and suppose (by way of contradiction) that a thread executing \( T \) can in fact race with a thread executing \( U \), 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 Fig. 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 Fig. 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, a practical, automated tool for detecting data races in drivers.

Architectures Fig. 7 depicts the WHOOP toolchain. The input to WHOOP is a Linux driver written in C, together with an environmental model [I] that is required to “close” the driver so that it can be analyzed for races. Initially, WHOOP uses three LLVM-based tools [31, 32], Chauffeur, Clang [33], and SMACK [34], [35], to translate the driver and its environmental model into an abstract program written in the Boogie intermediate verification language (IVL) [36]. Boogie is a simple imperative language with well-defined semantics that is used as the input to a number of cutting-edge verifiers (e.g. the Boogie verifier and CORRAL). Next, WHOOP instruments and sequentializes the program to perform symbolic pairwise lockset analysis (see Fig. 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 Fig. 7-C and [IV]).

We engineered the Chauffeur [I] and WHOOP [V] components of our toolchain (denoted with grey boxes in Fig. 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 functions. Linux drivers define entry points in a standard way (see Fig. 8 for an example of how the generic_nvram driver defines the entry points for file_operations); 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, 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.

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 considered to be an array of bytes [37]. 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 Boogie constant. Representing all locks statically, instead of using their original SMACK pointer-based representation, makes it easier for WHOOP to perform lockset instrumentation and lockset-based invariant generation. Currently, WHOOP supports Linux kernel mutex and spinlock operations; it is easy to enhance it with knowledge of other locking primitives. If WHOOP cannot reliably infer a unique lock instance (e.g. because it is stored in an unbounded data structure of locks such as an array or a list), it will exit with a warning. However, we have never encountered this in practice since 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 Fig. 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 \( \text{Bool} \), 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 [33] 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 previous work [39]. Suppose we are analyzing entry points \( T \) and \( U \), and that after translation into Boogie IVL these entry points share a common memory region \( MR \). When analyzing \( T \) and \( U \) for races, we introduce an unconstrained symbolic constant \( \text{watched}_{MR} \), 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 \( \text{watched}_{MR} \). If we can succeed in proving this, we know that \( T \) and \( U \) must be race-free for the whole \( MR \), 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, and that some drivers do exhibit recursion, e.g. the r8169 ethernet driver (see IV). To make our analysis scale while maintaining precision, and to support recursion, we use the Houdini algorithm [40] 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. This is done 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 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 \( m \notin \text{CLS}_i \) as a post-condition for \( f_i \); this guess may be incorrect, 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.

Verification and Error Reporting For each entry point pair, the instrumented sequential program, equipped with procedure and loop summaries, is given 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, implementing the sequential program sketched in Fig. 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 Fig. 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 ones.

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 IV): 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.

Second, we soundly reduce the number of memory regions that are analyzed for races. If memory region MR is accessed for races, then 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 “soundly”\footnote{\url{http://soundness.org/}}: 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.

In addition to inheriting soundness issues arising from currently unknown bugs in WHOOP and in the external tools that WHOOP relies on, we acknowledge that: (i) the 1.5.0 release of SMACK we used 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\footnote{Note that currently bit-vectors and unaligned byte-level memory accesses are soundly and precisely handled by SMACK (as of release 1.5.1).} 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 latter 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\footnote{\cite{single-open-device}} 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, 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, experts\footnote{\cite{false-alarms}} advise 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 be inaccurate, leading to semantic mismatches that can in turn lead to false positives and/or false negatives. However, we currently only focus on finding data races, and thus can get away with over-approximating large parts of the Linux kernel, without losing too much precision. Making our model more precise is an ongoing manual effort that requires Linux kernel 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\footnote{\cite{false-alarms}} and\footnote{\cite{false-alarms}}.

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\footnote{\cite{corral}}, a precise bug-finder used by Microsoft to analyze Windows drivers\footnote{\cite{windows-drivers}}. Because CORRAL operates on Boogie programs, it was easy to integrate it into our toolchain (see Fig.\footnote{\cite{corral}}). 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 \textit{sequentialized}, and the generated sequential version preserves reachable states of the original concurrent program up to the context bound\footnote{\cite{sequentialization}}. 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) 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\footnote{\cite{race-checking}}. Whenever there is a write access to a shared variable $s$, WHOOP instruments the program as follows:

\begin{verbatim}
 s = e; \textit{// original write}
yield; \textit{// allow for a context-switch}
assert s == e; \textit{// check written value}
\end{verbatim}
Likewise, whenever there is a read access to $s$, WHOOP instruments the program as follows:

```plaintext
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 races. Note that our instrumentation conveniently tolerates some benign races: it does not report a read-write race if the write access updates the shared memory location with the same value that already existed; it also does not report a write-write race if the two write accesses update the shared memory location with the same value (which can be different from the pre-existing 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. 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, and we also omit the `assert` that would check for a race. This is a form of [partial order reduction](#1), 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.

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 (EPP) 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 (MR) that might race in the pair (regardless if the pair has not been fully proven as race-free), and after each lock and unlock statement.

In our experiments (see [V]), Yield-MR is significantly faster than Yield-EPP.

Our partial-order reduction is able in principle to accelerate CORRAL for arbitrary bug-finding in concurrent programs. Although we did preliminary explorations in this direction, identifying useful safety properties to check was challenging since drivers typically contain no assertions. Thus, in this paper we use CORRAL solely to find data 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. 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.

#### 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 1.5.0, Z3 4.3.2, Boogie rev. 4192 and CORRAL rev. 534. We also used Mono 4.1.0 to run Boogie and CORRAL. We configured CORRAL with a time budget of 10 hours (T.O. denotes timeout), a context-switch bound (csb) of 2, 5, and 9, and the default recursion depth bound of 1.

#### 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 our benchmarks: lines of code (LoC); number of possible 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 data races discovered by CORRAL using a csb of 2 (#Races Found). Using a higher csb than 2 did not uncover any further races; this might mean that all races in our benchmarks can manifest with a csb of 2, or that CORRAL hit its default recursion depth bound of 1 before discovering a deeper bug. Although we experimented with higher recursion depth bounds, we were unable to discover any races that could not be exposed with the default recursion depth bound.

We can see in Table [I] that WHOOP reports more races than CORRAL does. This is because WHOOP employs an over-approximating shared state abstraction to conservatively model

---

1. 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.

---

1. [https://www.kernel.org](https://www.kernel.org)

1. The number of racy memory regions can be less than the number of races found by CORRAL: WHOOP might find that a memory region is racy, but the same memory region might race in several program statements.
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 previous work [42]. This can potentially lead CORRAL to report false bugs, but we have not seen this in practice. Furthermore, 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. Note that standalone 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.

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. 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 Fig. 1): the lseek entry point accesses the file offset file->f_pos without holding a lock (file is passed as a parameter to lseek). This causes a race if the kernel invokes lseek from another thread, while passing the same file object as a parameter. We observed that another char driver, using the same APIs, does use a lock to protect the offset access in its respective lseek entry point, leading us to suspect that the race we found in generic_nvram is not benign.

**Accelerating CORRAL** Table I presents runtime results from using WHOOP, standalone CORRAL, and WHOOP + CORRAL to analyze our benchmarks, while Fig. 9 plots the runtime speedups that CORRAL achieves using WHOOP with the Yield-MR instrumentation. 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). All reported times are in seconds and averaged over three runs.

WHOOP uses over-approximation to scale 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 same driver in 144.5 seconds. We believe that the reason behind this is that the r8169 driver has deeply-nested recursion in some of its entry points, which puts burden on CORRAL’s stratified inlining. This is not an issue for WHOOP, which uses procedure summarization. This shows that WHOOP has value as a standalone 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. Fig. 9 shows that most speedups using Yield-MR are between 1.5× and 10×; in ssu100 and pc8736x_gpio with a csb of 9 we achieved a speedup of 12× and 20×, respectively. We noticed that a higher csb typically results in greater speedups when exploiting WHOOP. This is expected as complexity grows exponentially with csb, and hence WHOOP’s reduction helps more at a higher csb. However, there are cases where WHOOP might slow down CORRAL. We noticed this in the sx8 driver: WHOOP verified 46 out of 47 memory regions, but did not fully verify any of the pairs; CORRAL, on the other hand, analyzed the only two pairs of the driver in just 21.4 seconds (csb of 9).

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

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

Fig. 9. Scatter plot showing the runtime speedups that CORRAL achieves using WHOOP with the Yield-MR instrumentation. The symbols +, o, and × represent a context-switch bound of 2, 5, and 9, respectively.
We believe that in this case the overhead of running WHOOP outweighed the benefits of using Yield-EPP or 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); we also could not find source code or binaries of other tools [48], [49], [50].

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 [51]. Warlock [52] and LockLint [53] 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, hinting at 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.

Choi et al. [54] combine static analysis and runtime access caching to speed up dynamic race detection. Kahlon et al. [48] use a divide-and-conquer algorithm that partitions all pointers of a program that do not alias in disjoint sets to achieve scalability; more recently, they used abstract interpretation to achieve a sound partial-order reduction on the set of thread interleavings and statically reduce the number of false race warnings [49]. Das et al. [50] employ 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. WHOOP uses symbolic lockset analysis, which involves generating verification conditions and discharging them to a theorem prover, and then employs CORRAL to filter out false races.

Our pairwise approach to analyzing 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 [55], [59]. The idea of pairwise analysis of components in a concurrent system has been broadly applied, notably in model checking of cache coherence protocols [56].

VII. CONCLUSIONS

In this paper we presented WHOOP, a new automated 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 scales 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 concurrency bug-finder.

ACKNOWLEDGMENTS

We thank Akash Lal for his input and for helping us with CORRAL issues, and Montgomery Carter for modeling Linux locks. Thanks to Jeroen Ketema, Tyler Sorensen, Anton Burtsev, and Charlie Jacobsen for discussions and feedback in various stages of this work. This work is part of the project “Automatic Synthesis of High-Assurance Device Drivers” and was generously funded by a gift from Intel Corporation; it was also supported in part by NSF CCF 1346756. A sketch of the WHOOP approach was presented at the 2014 Imperial College Computing Student Workshop [57].