Verifying Absence of Hardware-Software Data Races using Counting Abstraction

Tuba Yavuz
ECE Department
University of Florida
Gainesville, FL, USA
tuba@ece.ufl.edu

***Abstract*—Device drivers are critical components of operating systems. However, due to their interactions with the hardware and being embedded in complex programming models implemented by the operating system, ensuring reliability of device drivers remains to be a challenge. In this paper, we focus on the interaction of the driver with the device and present an approach for modeling this interaction and verifying absence of hardware-software data races. Specifically, we use the counting abstraction technique to abstract dynamic process creation in response to I/O acknowledgements sent by the device. We present the results of our approach on the modeling and verification of several Linux device driver models.**

I. Introduction

Device drivers are critical components of operating systems due to their role in interfacing with the hardware. Operating systems are complex software systems that are designed to meet various goals such as extensibility, reliability, high-performance, and security. This yields a complicated programming model in which the device drivers are subject to operate. In addition to the rules of the programming model that common kernel components are subject to, each driver also needs to conform to a programming model that is imposed by the underlying communication bus, e.g., USB, PCI, etc.

A recent study by Google [1] reports that kernel bugs are on the rise and that %85 of the kernel bugs reported for Android are within the driver code. Despite recent advances in the decision procedures and program analysis, thorough analysis of device driver code remains to be a challenge due to complexity of device drivers. On the other hand, lack of modeling languages that can support model-driven development of device drivers is preventing early detection of bugs during development.

In this paper, we introduce a modeling and verification approach for verifying absence of hardware-software (hw-sw) data races. We provide a modeling approach that does not require a detailed model of the device for reasoning about absence of hw-sw races. We use the counting abstraction technique [2] to model potentially unbounded number of completion handler instances and other driver threads that get dynamically created. We have implemented our approach in a tool, called RIOT, and performed case studies on several Linux USB drivers. Our preliminary results suggest that absence of hw-sw races can be proved on the models of Linux USB device drivers.

978-1-7281-9148-5/20/$31.00 © 2020 IEEE

The rest of the paper is organized as follows. In Section II, we introduce a motivating example for the complexities involved in device-driver interactions. In Section III, we introduce our modeling and verification approach that uses the counting abstraction technique. In Section IV, we explain our implementation and present several case studies. Section V compares our approach to the related work. Finally, Section VI concludes with directions for future work.

II. Motivation

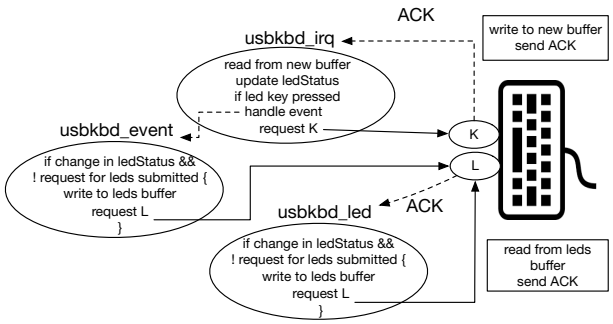

Fig. 1: Interaction between the driver and the keyboard.

We consider a programming model that involves three types of processes: the server, the client, and the environment. A server process, S, receives generic requests for input (output) over some shared object m and responds by performing a write (read) operation on m. The completion of a request is signaled through an acknowledgement, which causes the dynamic creation of a client process, C. Upon creation, the client process accesses the shared object m for read and/or write operations and terminates. The environment processes model processes other than the client and the server processes. We assume that all processes run concurrently. A process is dynamic if it can be created and is static otherwise. Note that a dynamic process may have potentially an unbounded number of instances. A process is parameterized if there are unbounded number of static instances for that process. A process is a single instance if it is neither dynamic nor parameterized. The environment may consist of parameterized, dynamic, and single instance processes.

An example instantiation of this programming model occurs in the Linux `usbkbd` driver [3] as shown in Figure 1. The keyboard device defines two endpoints or I/O ports: 1) The `K` endpoint represents requests for reporting of key events in the last window, 2) The `L` endpoint represents a control endpoint that can be used to send the led commands (turn on/off) if an Led related key, e.g., CAPSLOCK, has been pressed in the last window. When the device gets a `K` type request, it fills the `new` buffer with the keys held down in the last window and finally responds by sending an acknowledgement that causes creation of a thread that runs the relevant completion handler, e.g., the `usbkbd_irq` function. The first `K` type request gets submitted by the `usbkbd_open` function (not shown in Figure 1). All other `K` type requests get submitted by `usbkbd_irq`.

When the device gets an `L` type request, it reads the `leds` buffer and updates the LEDs on the keyboard accordingly. It also sends an acknowledgement to the driver, which causes the creation of a thread that executes the `usbkbd_led` function. `L` type requests can be submitted from two different threads that run the `usbkbd_event` and the `usbkbd_led` functions, respectively. In the `usbkbd_irq` function, the driver calls an API function of the input layer, `input_report_key`, to pass key events to the input layer. For LED related keys, this function causes the creation of a thread that executes the `usbkbd_event` callback of the driver. If there has been a change in the led key presses and there are no `L` type requests in flight, i.e., submitted but not acknowledged yet, then it updates the `leds` buffer to reflect which LEDs should be turned on or off and submits an `L` type request. When a thread that runs the `usbkbd_led` function gets created upon receipt of the acknowledgement, it also checks if there is a need to submit an `L` type request and whether there are no requests in flight at the time. If so, it updates the `leds` buffer and submits an `L` type request. It should be noted that Figure 1 abstracts a lot of the details.

A key property for the correct functioning of the driver is the absence of data races between the driver and the keyboard. So, the driver and the device should not access the shared buffers `new` and `leds` at the same time and must employ proper synchronization. Otherwise, hazards such as loss of key events may happen. For instance, the device may overwrite the `new` buffer before the driver gets a chance to read the key events from the previous window or the driver may overwrite the `leds` buffer before the device gets a chance to read the commands for the previous window.

Synchronization primitives that are used for mutually exclusive access to critical regions cannot help to prevent hardware-software data races. Although the device can access the shared memory (via Direct Memory Access (DMA) or exposing its I/O ports), it cannot execute these synchronization primitives on the host CPU. So, it is the responsibility of driver developers to realize a protocol for ensuring mutually exclusive access to the shared memory. The above programming model helps achieve such a protocol by accessing the memory shared with the device only inside the completion handlers, which executes after the device acknowledging the end of its access. However, as it can be seen in Figure 1, it is possible to submit multiple I/O requests from different contexts, `usbkbd_event` and `usbkbd_led`, if software data races exist between these two threads. So, to reason about absence of hardware-software data races, we need to analyze the driver with all its components including the dynamically created threads.

III. Approach

We present our modeling, abstraction, and verification approaches for reasoning about hardware-software (hw-sw) data races in Sections III-A, III-B, and III-C, respectively.

A. Modeling the Device-Driver Interaction

We model the interaction of the driver and the device using three types of processes: the environment, the server, and the completion handlers. An environment process models an entry point of a driver such as those accessed by the file system layer (VFS) as in the Linux kernel, e.g., `usblp_write` entry point for the `usblp` driver, or an entry point from other kernel layers such as the `usbkbd_event` callback instantiated by the Input layer for the `usbkbd` driver. In Figure 2, `event`, `keyboard`, and `irq&led`, represent the environment process, the server process, and the completion handlers, respectively.

The server process is for modeling handling of the I/O requests by the device, i.e., read and/or write actions performed by the device on the relevant buffers and responding to the host by dynamically creating an instance of the relevant completion handler. Since each I/O request is associated with a specific endpoint e, we model each endpoint by associating it with two integer variables, req_e and ack_e, which represent the number of requests received and the number of acknowledged requests, respectively. These variables are typically initialized to 0. However, it is also possible to start the system from a state where some requests have been made, e.g., $req\_K == 1$ in the initial state of the `usbkbd` driver abstracting away the `usbkbd_open` function, which submits the initial request for keyboard events.

An I/O request from the driver through the endpoint e is modeled by incrementing req_e by one in the process performing the I/O request. Once the endpoint e performs the actual I/O, it increments ack_e by one and creates an instance of the completion handler. Assuming that `buf_e` and `comp_e` represent the I/O buffer and the completion handler associated with endpoint e, respectively, a typical server process works as follows:

```
1  check_request:
2              if (req_e > ack_e)
3                 read/write buf_e
4                 ack_e++
5                 create comp_e
6               else
7                 goto check_request
```

Instead of representing the content of the I/O buffer and the updates explicitly, we model it using predicates, e.g., $ledKey$ as shown on line 14 in Figure 2. $ledKey$ represents whether a led key is reported in the `new` buffer in Figure 1. So our

```
1  module usbkbd
2    leds_lock: lock;
3    requestIrq, ackIrq: int;
4    changeInLed,  ledSubmitted, ledKey: bool;
5    initial req_K == 1 && ack_K == 0;
6    restrict req_K >= 0  && ack_K >= 0;
7    initial req_L == 0 && ack_L == 0;
8    restrict req_L >= 0 && ack_L >= 0;
9    # syntax for absence of hw-sw data race property:
10   # ["i"|"o"]"hwswrace" device, req, ack, sharedPred;
11   ihwswrace keyboard, requestIrq, ackIrq, ledKey;
12   ...
13   proc irq()
14     if (ledKey == True) {
15        changeInLed := True;
16        create event;
17     }
18     req_K := req_K + 1;
19   endproc
20
21   proc event()
22     atomic leds_lock {
23        if (ledSubmitted == False) {
24           if (changeInLed == True) {
25              changeInLed := False;
26              req_L := req_L + 1;
27              ledSubmitted := True;
28    } } }
29   endproc
30
31   proc led()
32     atomic leds_lock {
33        if (changeInLed == False) {
34           ledSubmitted := False;
35        }
36        if (changeInLed == True) {
37           changeInLed := False;
38           req_L := req_L + 1;
39       } }
40   endproc
41
42   proc keyboard()
43      while (True) {
44         if (req_K > ack_K) {
45            ledKey := *;
46            ack_K := ack_K + 1 ^ create irq;
47         }
48         if (req_L > ack_L) {
49            ack_L := ack_L + 1 ^ create led;
50     } }
51   endproc
52 endmodule
```

Fig. 2: A model of the `usbkbd` driver, Linux version 4.12, as specified using the input language of RIOT.

server model replaces a write operation as on line 3 in the above code snippet with non-deterministic updates to such predicates, denoted by `pred=`$\star$, as shown on line 45 of Figure 2. Although we can use the program locations of the server that update such predicates to refer to the states in which the device has access to the I/O buffers, we will be utilizing the constraint `req_e > ack_e` in the specification of hw-sw race conditions to refer to such states.

The completion handlers can be potentially unbounded due to potentially unbounded number of I/O requests, which, in turn, leads to the creation of completion handlers with multiple concurrently running instances. As an example, submission of the I/O requests for the led endpoint shown in Figure 1 are modeled by the increment operations on lines 26 and 38 in Figure 2. Some of the environment processes, such as the `usbkbd_event` thread in the `usbkbd` driver, may also be unbounded due to dynamic thread creation as modeled by line 16 in Figure 2. Finally, the correctness property for absence of hw-sw data races involving the input buffers is specified as shown on line 11 in Figure 2.

B. Formalizing the Device and Driver Interaction

Given the fact that there can be potentially an unbounded number of I/O requests in a device driver model, the interaction between a device and the driver needs to be modeled as an infinite-state transition system. We can represent each process type with a transition system and define the composite system as an asynchronous composition of the transition systems for the instances of each process type. However, the unbounded number of instances need to be represented using a finite encoding, which can be done using infinite state variables. Counter abstraction [2] technique achieves this in an elegant way and provides strong guarantees about the correctness of the original transition system for properties that involve global variables only. In what follows, we explain the semantics by using either the set theoretic or formula representation to simplify the presentation. We use the notation $[\![f]\!]$ to denote the set theoretic representation of a formula f.

Given a Kripke structure $T = (S, I, R)$, we define the counted abstracted version of T, denoted by $T^{\mathcal{CA}}$, in terms of three types of transformations. 1) Each local state is abstracted away by introducing a unique integer variable to count the number of instances of T in that local state, i.e., $T^{\mathcal{CA}}.[\![S]\!] \equiv \prod_{g \in V_G} Dom(g) \times \mathcal{Z}^n$, where G, $Dom(v)$, and n represent the set of global variables in T, the domain/type of a variable v, and the number of counter variables, respectively. 2) Assume that the set of initial states of T is defined as $I_G \wedge I_L$, where I_G and I_L define the initialization of the global and local variables, respectively. Initial states are rewritten by dropping the local variables and using the counter variables, i.e., $T^{\mathcal{CA}}.I \equiv I_G \wedge \bigwedge_{s \in [\![\prod_{l \in V_L} Dom(l)]\!]} CR(s) = 0$, where V_L and $CR(s)$ denote the set of local variables in T and the counter variable that maps to the local state s. So, we assume that there are no dynamic processes in the initial state. 3) Assume that $T.R \equiv \bigvee_{i=1}^{n} gg(r_i) \wedge lg(r_i) \wedge gu(r_i) \wedge lu(r_i)$, where $gg(r)$ and $lg(r)$ denote the guard expressions on the global and local variables for atomic transition r, respectively, and $gu(r)$ and $lu(r)$ denote the update expressions on the global and local variables for atomic transition r, respectively. Each atomic transition in T is transformed to drop the local variables and use the counter variables in the guard and update parts of the transition formula.

$$T^{\mathcal{CA}}.R \equiv \bigvee_{i=1}^{n} gg(r_i) \wedge gu(r_i) \wedge \Big(\bigvee_{s \in [\![lg(r_i)]\!]} CR(s) > 0 \Big) \wedge \bigvee_{s_1 \in [\![lg(r_i)]\!], s_2 \in sl(r_i)} \Big(CA(s_1, s_2) \wedge \bigwedge_{s \notin \{s_1, s_2\}} CR'(s) = CR(s) \Big) \tag{1}$$

and

- $sl(r)$ denotes the local states that can be reached in one step with transition r, i.e.,

$$sl(r) \equiv [\![\exists V_G.\mathcal{POST}(true, r)]\!]$$

, where V_G is the set of global variables in T.

- $CA(s_1, s_2)$ represents the equation that simulates the transition from local state s_1 to local state s_2:

$$CA(s_1,s_2) = \begin{cases} CR(s_1)' = CR(s_1) & s_1 = s_2 \\ CR(s_2)' = CR(s_2) + 1 & s_1 \neq s_2 \\ \wedge\ CR(s_1)' = CR(s_1) - 1) & \end{cases}$$

Finally, we model the effect of dynamic creation of a process defined by the transition system T using the atomic transition $(\bigvee_{s \in [\![T.I_L]\!]} CR'(s) = CR(s) + 1) \wedge \bigwedge_{s \notin [\![T.I_L]\!]} CR'(s) = CR(s)$.

In our modeling of the device and driver interaction, we transform each dynamically created process using the counting abstraction transformation as explained above. So, for the `usbkbd` driver shown in Figure 2, we generate the Kripke structures for the potentially unbounded number of instances of the processes `irq`, `event`, and `led` using counting abstraction and represent the single instance of the keyboard using the original Kripke structure. Note that the local variables for the model in Figure 2 are implicit and consist of the program counters of the dynamic processes.

Let the atomic transition $pc = 15 \wedge changeInLed' \wedge pc' = 16$ denote the execution of the statement on line 15 in Figure 2 and let pc denote the program counter for the `irq` process and primed variables denote the next state variables. Let c_{15} and c_{16} denote the counter variables introduced to represent the number of instances of the `irq` processes with their program counters being at lines 15 and 16, respectively. The counter abstracted version of this transition is $c_{15} > 0 \wedge changeInLed' \wedge c'_{15} = c_{15} - 1 \wedge c'_{16} = c'_{16} + 1$.

Definition 1 *An atomic predicate is called* counting-abstraction preserving *(CAP) in the context of a composite transition system with counter abstracted components if it does not involve any local variables of the counting abstracted processes and it involves the counter variables in the form of the guard formulas, i.e., $CR(s) > 0$, that are generated as part of the counting abstraction transformation of atomic transitions.*

Theorem 1 *There exists a simulation relation between a composite transition system with counter abstracted components and the original composite transition system with dynamic components with respect to a state labelling function on any predicate set that contains only counting abstraction preserving predicates.*

The proof for Theorem 1 follows from our construction of $T^{\mathcal{CA}}$ and from Definition 1. The simulation relation for predicates that involve the global variables follows from the bisimulation equivalence with respect to the global variables. The predicates that involve the counter variables yield a simulation relation due to being in restricted form and satisfying $[\![f]\!] \subseteq \gamma(\alpha([\![f]\!]))$, where α is the counting abstraction of a set of states represented by formula f and γ is the mapping of a state described in terms of the counter variables to a state described in terms of the local variables of the original transition system.

Algorithm 1 An algorithm for checking absence of hw-sw data races for a given dynamic composite transition system, T, a specific I/O port, req and ack, the direction of I/O, $isInput$, and the shared buffers, $shared$.

1: **CheckHWSWRaces**($T = (Env, Server, Client)$: Composite Trans. Sys., req: V, ack: V, $isInput$: Boolean, $shared$: 2^V)
2: $T^{\mathcal{CA}} = (S, I, R) \leftarrow CountingAbstract(T)$
3: $prop \leftarrow False$
4: **for** each $[\![r]\!] \in [\![R]\!]$ not executed by processes in $Server$ **do**
5: Let $r \equiv cg(r) \wedge dg(r) \wedge du(r) \wedge cu(r)$
6: **for** each v in $shared$ **do**
7: **if** $isInput$ and $v \in Var(dg(r))$ **then** ▷ Identify read accesses
8: $prop \leftarrow prop \vee cg(r)$ ▷ Record the control guard
9: **end if**
10: **if** $v \in Var(du(r))$ **then** ▷ Identify write accesses
11: $prop \leftarrow prop \vee (cg(r) \wedge dg(r))$ ▷ Record the control and data guard
12: **end if**
13: **end for**
14: **end for**
15: $prop \leftarrow \neg(prop \wedge req > ack)$
16: **return** $T^{\mathcal{CA}} \models prop$

C. Checking Absence of HW-SW Races

A data race is a condition when two threads access a shared variable in a way that at least one of the accesses is a write and there is no proper synchronization between the two threads, i.e., the result depends on who executes first.

In a hw-sw race, the two threads run in different execution environments: one inside the driver and the other on the device. In the context of our modeling approach, each type of thread is modeled as a process. For checking the absence of hw-sw races, it is sufficient to have an abstract model of the device that 1) creates a completion process when there is a new I/O request and 2) nondeterministically updates any control relevant predicate on the shared variables.

To specify the absence of a hw-sw race as a safety property, we should also specify the states in which the driver is accessing the shared buffers. Here, we need to incorporate the direction of I/O as it determines which accesses are conflicting. If the request is an input request, i.e., the device is writing to the shared buffers, then both the read and the write accesses performed by the driver would yield a data race. However, if the request is an output request, i.e., the device reads the shared buffers, then only the write accesses performed by the driver constitute a data race.

Algorithm 1 takes as input a transition system, T, a specific I/O port represented by the pair of I/O requests, req, and completed I/O requests, ack, the direction of I/O, $isInput$, and the set of shared buffers/predicates over such I/O requests. It first generates a counting abstracted [2] version, $T^{\mathcal{CA}}$, of the transition system (line 2). Then, it constructs the safety property $prop$ that specifies the absence of hw-sw races on the given I/O port and the shared buffers.

We assume that each atomic transition, r, in $T^{\mathcal{CA}}$ can be represented by a conjunction of four formulas: the guards on the control variables, $cg(r)$, and the global variables, $dg(r)$, and the updates on the global variables, $du(r)$, and the control variables, $cu(r)$. For processes that are not dynamically

created in T and, hence, are not replaced by their counting abstracted versions in $T^{\mathcal{CA}}$, $cg(r)$ corresponds to $lg(r) \wedge gg(r)$. For counting abstracted processes, $cg(r)$ corresponds to a formula of the form $c > 0$, where c represents the counter variable for the local state of the process at which r is enabled.

So, the algorithm analyzes the guard and the update of each atomic transition executed by any non-server process (line 4), i.e., the driver threads, to identify conflicting operations by the driver for a possible hw-sw data race[1]. As we mentioned above, the read accesses by the driver are relevant for input type requests only. In such cases, read accesses are identified for any shared variable that appears on the guard on the global variables. Write accesses are relevant for both types of I/O requests and are identified for any shared variable that appears on the part of the transition that updates the global variables.

For conflicting read accesses, the constructed property includes the constraint for the control guard (line 8) as this is sufficient for the driver to read the variables in $dg(r)$, even though $dg(r)$ may turn out to be false, in which case the update part of the transition would not take effect. We assume that the data guards that check the availability of mutual exclusion locks are not mixed with the guards on other types of global variables. More details about our system construction are provided in Section IV-A. However, for the write accesses to be feasible both the control guard and the guard on the global variables need to hold and, hence, the property includes their conjunction (line 11). An example formula for the hw-sw data race absence property regarding the shared variable, $ledKey$, related to the `K` endpoint (input) from Figure 1 is $\neg(c_{14} > 0 \wedge req\_K > ack\_K)$, where c_{14} represents the counter that was introduced to represent the program counter of the `irq` process referring to the `if` statement on line 14 in Figure 2. The property regarding the shared variable, $changeInLed$, related to the `L` endpoint (output) from Figure 1 is $\neg(((c_{14} > 0 \wedge ledKey == True) \vee (c_{24} > 0 \wedge changeInLed == True) \vee (c_{36} > 0 \wedge changeInLed == True)) \wedge req\_L > ack\_L)$, where c_{24} and c_{36} are the counters for the program counters of the `event` and `led` processes referring to the `if` statements at lines 24 and 36 in Figure 2, respectively. Note that the model does not show that the shared predicate $changeInLed$ get explicitly accessed, i.e., read, by the `Keyboard` process as we abstract away the side-effect of turning on/off the leds in our model. This does not affect the correctness of the constructed property formula as the server processes are not used to derive the property as indicated in line 4 of Algorithm 1.

Since hw-sw races can be specified using counting-abstraction preserving predicates, the soundness of our verification approach follows from Theorem 1.

IV. Preliminary Results

A. Implementation

We have implemented our approach for detecting hw-sw data races on the counting abstracted versions of dynamic composite systems in a tool called RIOT[2] We use a simple while-language with additional features: 1) atomic blocks that specify a mutual exclusion lock name to model blocks that are enclosed between the acquire and release operations on the specified lock, 2) parallel assignment and dynamic process creation using the ˆ operator, 3) nondeterministic assignment for boolean variables using the $\star$ operator, 4) proc declarations modeling a process/thread, 5) dynamic process creation using the `create` operator, and 6) specifications for the initial states, state space restrictions, and correctness properties including the hw-sw race freedom. Each process is interpreted as a modular transition system and the system behavior is described in terms of their asynchronous composition. Each basic statement and conditional checks are interpreted as atomic transitions, e.g., an if statement such as `if (a) { b := True }` translates to a single transition while `if (a) { b := True; c := True; }` translates to multiple transitions. Each atomic transition is identified with a unique location and each process instance except those that are dynamically created is assigned its own program counter.

Driver	$\|C\|$	$\|U\|$	Property	R	Time (secs)
usbkbd	2	3	I, {ledKey}	✓	27.09
			O, {changeInLed}	✓	6.95
usblp	2	4	I, {wactual_length}	✓	14.95
			O, {writebuf}	✓	211.71
			I, {ractual_length}	-	> 3753.75
onetouch	1	1	I, {data}	✓	1.62
airspy	1	1	I, {buf_list[i]}	✓	30.85
tiusb	3	5	I, {transf_buf1}	✓	13.39
			I, {trans_buf2, actual_length}	-	> 4901.77
			O, {trans_buf3}	✓	5694.74
legousb	2	4	I, {in_urb}	✓	1500.27
			I, {actual_length}	-	> 2362.47
			O, {out_urb}	-	> 6748.29

TABLE I: Verification results for checking the absence of hw-sw data races in Linux USB driver models. $|C|$ and $|U|$ represent the number of I/O completion processes and dynamically created processes, respectively. Property specifies the type of I/O (I or O) and the shared buffer (predicates). R and Time presents the verification result and time in secs, respectively. – means the analysis could not finish due to the verification engine running out of memory.

Our tool automatically translates the transition system into its counted abstracted version and then encodes it into Horn clauses. Given a hw-sw race property specification that consists of the server process (the device), the request and acknowledgement counters, and the shared buffers/predicates, it analyzes the transition system as in Algorithm 1 to generate a safety property and uses the fixedpoint engine of the Z3 SMT solver [4] to check reachability of a hw-sw race. We use the monolithic translation of transition systems into Horn clauses as explained in [5]. Although our tool can use any of the

[1]We assume that each server process represents a device and each port on every device works on a mutually exclusive set of I/O buffers.

[2]The tool (the compiler for the simple while language, the counting abstraction transformer, and the verification engine that uses Z3) and the benchmarks can be found at https://drive.google.com/file/d/1wmeEVDOlQDyK63IQ0Zh9et9H1MmNjjXk/view?usp=sharing.

backend solvers of Z3 that support linear integer arithmetic, we used Spacer [6] in our experiments. We ran our experiments on an Intel Xeon 3.20GHz CPU with an 8GB RAM. Table I shows that verification time takes longer as the number of dynamically created processes increase.

V. RelatedWork

A formalization for hw-sw interfaces is presented in [7]. Similar to our approach, their formalization allows composition of driver and device models at a customizable abstraction level. Our approach focuses on the specific problem of hw-sw race detection. We use counting abstraction to transform the combined system model which may potentially involve an unbounded number of completion handlers/interrupt service routines (ISRs).

In [8] virtual device prototypes are analyzed using symbolic execution and conformance checking for the driver and the device is performed to detect DMA interface bugs. Our approach can pinpoint hw-sw races that may not involve DMA. Guardrail [9] monitors execution of a driver at runtime to detect hw-sw data races due to DMA faults. This work is complementary to our approach as it analyzes the original driver code. However, it can only detect bugs that are manifested during execution. Our model-based approach can provide formal guarantees as long as the model faithfully represents the driver-device interactions.

In [10] a fixed number of parallel thread instances are modeled using on-the-fly counting abstraction to tame the local state explosion problem. The BFC tool [11] uses counting-abstraction to represent multi-threaded software models with an unbounded number of threads with finite local states. While BFC leverages search algorithms for solving the coverability problem [12], our approach handles multiple types of unbounded processes using interpolation-based fixpoint solvers.

In [13] a modular approach for generating local invariants of a multi-threaded program is presented. The program is represented in terms of a data-flow graph that encodes the data flows between the statements in a program. The reachable states are discovered in an iterative fashion and a separate interference analysis is guided by the discovered incomplete invariants. User provided assertions can be checked when the iterative process terminates. In our approach, we automatically generate the property to be checked based on the user input that describes the shared buffer, the direction of I/O, and the specific port. Our approach is property directed and it avoids computing an invariant for each program location.

Counting abstraction has been applied to the verification of Smack models for the absence of software data races in [14]. In this paper, we explicitly model the interaction with the device while also modeling multiple driver threads.

VI. Conclusion

In this paper, we have presented a modeling and verification approach for checking the absence of hardware-software data races. We have shown the effectiveness of our approach on the verification of several Linux USB driver models. Our approach does not require a detailed hardware model. However, once verified for the absence of hardware-software data races, the model can be refined to incorporate more detailed models of hardware and driver components. In future work, we would like to explore semi-automatic generation of device driver models and optimization of the verification performance.

VII. Acknowledgements

This work was partially funded by the US National Science Foundation under grant CNS-1942235.

References

[1] J. V. Stoep, "Android: protecting the kernel," In Linux Security Summit. Linux Foundation, 2016.

[2] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, no. 3, Jul. 1992.

[3] "Linux usbkbd driver," https://elixir.bootlin.com/linux/v4.12/source/drivers/hid/usbhid/usbkbd.c.

[4] K. Hoder, N. Bjørner, and L. M. de Moura, "μZ- an efficient engine for fixed points with constraints," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 457–462.

[5] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A constraint-based verifier for multi-threaded programs," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 412–417.

[6] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," *Formal Methods in System Design*, vol. 48, no. 3, pp. 175–205, 2016.

[7] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey, "Formalizing hardware/software interface specifications," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, 2011, pp. 143–152.

[8] L. Lei, K. Cong, Z. Yang, and F. Xie, "Validating direct memory access interfaces with conformance checking," in *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, 2014, pp. 9–16.

[9] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Guardrail: A high fidelity approach to protecting hardware devices from buggy drivers," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014.

[10] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Symbolic counter abstraction for concurrent software," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, 2009, pp. 64–78.

[11] "Bfc - A Widening Approach to Multi-Threaded Program Verification," http://www.cprover.org/bfc/.

[12] A. Kaiser, D. Kroening, and T. Wahl, "A widening approach to multithreaded program verification," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, pp. 14:1–14:29, 2014.

[13] A. Farzan and Z. Kincaid, "Verification of Parameterized Concurrent Programs by Modular Reasoning about Data and Control," ser. POPL '12, 2012, p. 297–308.

[14] F. Fowze and T. Yavuz, "Specification, verification, and synthesis using extended state machines with callbacks," in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016*, 2016, pp. 95–104.