Effective Verification of Low-Level Software with Nested Interrupts

Daniel Kroening*, Lihao Liang*, Tom Melham*, Peter Schrammel*, Michael Tautschnig†
*University of Oxford, UK †Queen Mary, University of London, UK

***Abstract*—Interrupt-driven software is difficult to test and debug, especially when interrupts can be nested and subject to priorities. Interrupts can arrive at arbitrary times, leading to an explosion in the number of cases to be considered. We present a new formal approach to verifying interrupt-driven software based on symbolic execution. The approach leverages recent advances in the encoding of the execution traces of interacting, concurrent threads. We assess the performance of our method on benchmarks drawn from embedded systems code and device drivers, and experimentally compare it to conventional formal approaches that use source-to-source transformations. Our experimental results show that our method significantly outperforms conventional techniques. To the best of our knowledge, our technique is the first to demonstrate effective formal verification of low-level embedded software with nested interrupts.**

I. Introduction

Interrupts are a key design primitive for embedded software that interacts closely with hardware. The interrupt mechanism enables timely response to outside stimuli in a power-efficient way. Interrupts are commonplace in all styles of computing platforms, including safety-critical embedded software, low-power mobile platforms and high-end information systems.

But interrupt-driven code is difficult to engineer. Device drivers, typical examples of software that uses interrupts heavily, are known as the "fault-hotspots" of the Linux Kernel [4], [10]. The root cause of the problem is the nondeterminism inherent in systems that use interrupts; the hardware can divert control to the interrupt service routine (ISR) at any point in time, resulting in possibly surprising interactions between the code that is interrupted and the ISR. The problem is exacerbated by *interrupt nesting*, where the ISR itself can be preempted by interrupts with higher priority.

Most existing approaches to validating interrupt-driven software rely on testing. But testing is particularly ineffective in the case of nested interrupts, as the number of possible interleavings—i.e. interspersions of interrupts within a run of the code—grows exponentially in the number of interrupts that occur. Bugs are therefore easily missed, and any errors that are observed are difficult to reproduce.

The concurrent nature of interrupt generation and handling and the sheer size of the search space suggest a formal approach to validation. Model Checking, in particular, shines when applied to problems with many subtle corner cases. Several model-checking based approaches to the problem have therefore been explored. A standard technique is to *instrument* the source code of the program with calls to the interrupt procedure, in effect mimicking the semantics of the hardware interrupt. The instrumented program is then passed to a conventional analyser for sequential programs. The approach is straightforward to implement, and is in principle viable in the case of nested interrupts as well. Sophisticated instances of this type of approach use over-approximating analyses, akin to partial-order reduction, to determine program locations where the call to the ISR can be omitted safely—for example when variables not shared with the ISR are read or written [12], [3].

The interleaving semantics of programs with nested interrupts strongly resembles that of programs with multiple threads. In particular, it is clear that the behaviours under thread semantics are a superset of the interrupt semantics. This suggests an analysis using Model Checkers for concurrent programs, such as SPIN. False alarms caused by the overapproximation can be addressed by means of suitable instrumentation. We discuss this approach in Sec. IV-B.

The key contribution of this paper is a third, alternative approach. It leverages recent advances in SAT-based analysis of concurrent systems. The key idea is to collect the set of possible events (reads or writes) in the system and construct a formula that relates their relative ordering by means of a symbolic *happens-before* relation. The formula is then passed to a modern SAT or SMT-solver together with a constraint that specifies the set of error states. If the solver determines the formula to be satisfiable, an error trace giving the exact sequence of events can be extracted from the satisfying assignment.

The method has been shown to be effective for interleaving semantics of threads, including the case of multi-core memory models with relaxed consistency [1]. CBMC, a tool implementing this idea, was awarded the Gold medal in the "Overall" category in the 2014 software verification competition. The work on memory models reported in [1] explores a weakening of the thread-semantics to capture the additional behaviours in the multi-core case. By contrast, our contribution in this paper is to employ a specially designed *strengthening* of the interleaving semantics of threads—in the form of a symbolic encoding by a partial order that precisely captures the subset of interleavings required to model nested interrupts faithfully.

This novel encoding provides the first method for effective formal analysis of software with nested interrupts.

II. Nested Interrupts

Interrupt semantics is very platform-dependent. On some microcontrollers (e.g. ARM Cortex-M, AT89Cxx), priorities or priority groups can be configured, whereas others use fixed priorities (e.g. AVR1305). The primary way to implement priorities is by enabling and disabling interrupts appropriately, e.g. by means of API functions such as `sei` and `cli`.

In this paper, we use the following interrupt semantics: when entering an ISR, all interrupts are disabled by the hardware. The ISRs can re-enable specific interrupts by issuing

978-3-9815370-4-8/DATE15/©2015 EDAA

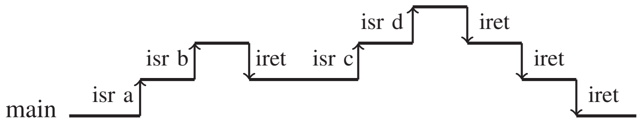
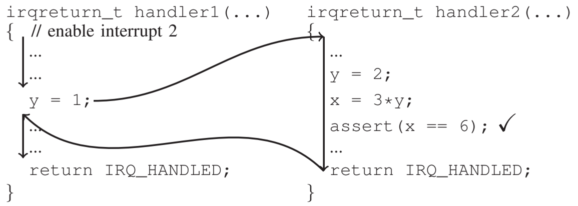


Fig. 1: A complex scenario of nested interrupts

a command, e.g. setting particular bits in a control register to one. The re-enabled interrupts can subsequently preempt the execution of the ISR that has re-enabled them.

Interrupts with priorities are a special case of this semantics, where the re-enabled interrupts are those with priority higher than that of the ISR that is executed. The handling of lower-priority interrupts is thus suspended until the handling of those with higher priority is completed. In the example from Fig. 1, interrupt a has a lower priority than b and c, and d has the highest priority. Hence, our semantics covers the vast majority of microcontrollers.

We illustrate the subtle semantics of programs with nested interrupts through the simple code fragment given in Fig. 2.



```
irqreturn_t handler1(...)     irqreturn_t handler2(...)
{  // enable interrupt 2      {
   ...                           ...
   ...                           y = 2;
   y = 1;                        x = 3*y;
   ...                           assert(x == 6); ✓
   ...                           ...
   return IRQ_HANDLED;           return IRQ_HANDLED;
}                             }
```

Fig. 2: A motivating example

We focus on verification of safety properties, e.g. user-defined assertions in interrupt-driven programs. If interrupt `handler1` re-enables interrupt 2 and interrupt `handler2` does not re-enable any interrupts, then the property `assert(x == 6)` in `handler2` always holds—because the execution of `handler2` cannot be interrupted. But an interleaving semantics of threads would admit the possibility that `assert(x == 6)` is violated. So tools for multi-threaded software verification, naïvely applied, can report an assertion violation that is not possible in any actual execution of the program. This motivates the need for techniques that model the semantics of interrupts more precisely than threads do.

III. Partial-order Encoding

This section presents the primary contribution of this paper: a new symbolic encoding for interrupt-driven programs with interrupt nesting. The idea is to translate a program into atomic memory read/write events, and then encode all interleavings of these events that can possibly occur as a symbolic partial order in a SAT/SMT formula. The encoding is a formula of the form $ssa \wedge pord$, where ssa encodes the data and control structure of the interrupt-driven program in the style expounded in [1], and $pord$ encodes all possible interleavings of the ISRs in the program. A satisfying assignment to the variables in the formula corresponds to an error trace in the interrupt-driven program.

We emphasise that our partial-order encoding for nested interrupts is not to be confused with the use of partial-order reduction [7], which uses total orders as starting point, and introduces partial orders to reduce the size of the state space.

We now discuss algorithms for computing the conjunct $pord$, which describes all possible interleavings between the ISRs. Algorithms 1–3 take as input a set of symbolic events of a program with nested interrupts and its program order po, which is the order in which program statements appear in each ISR. They produce as output sets of constraints that encode the four relations between symbolic events: *write serialisation* (ws), *read-from* (rf), *from-read* (fr) and *nested interrupts* (nested-isr), based on an extension of the framework by Alglave et al. [2]. Conceptionally, these relations can be seen as directed edges of an event graph, where an edge from a vertex s to a vertex t represents the assertion that memory event s globally happens before memory event t in an execution of the original interrupt-driven program. The union of po, ws, rf, fr, and nested-isr logically encodes a partial order over the memory events that arise in program execution. An execution represented by an event graph is *valid* if po $\bigcup$ ws $\bigcup$ rf $\bigcup$ fr $\bigcup$ nested-isr is acyclic, i.e., there are no cyclic dependencies among the read and write events in the event graph. The formula $pord$ is the conjunction of all constraints produced by Algorithms 1–3.

For every event e, we define an integer variable cl_e, which is the *clock* of e. Given two memory events e and e', event e happens before e' if it has a smaller clock value, i.e., if $cl_e < cl_{e'}$. We write $c_{ee'}$ as a shorthand for g(e) $\wedge$ g(e') $\Rightarrow$ $cl_e < cl_{e'}$, where g(e) is the guard of the SSA equation from which e is constructed. We write addr(e) = addr(e') to assert that e and e' access the same memory address, and tid(e) = tid(e') to assert that e and e' belong to the same *invocation* of an ISR. Note that the fact that e and e' belong to the same ISR does not necessarily imply tid(e) = tid(e'), as an interrupt can re-enable itself. We define enable(e, e') to be true if the ISR call that has a memory event e enables the interrupt whose ISR has a memory event e'. We write enables for the transitive closure of enable. In the special case where interrupts have explicit priorities, enable(e, e') would mean that the priority of the interrupt whose ISR has a memory event e is lower than the priority of the interrupt whose ISR contains e'.

Recall that when entering an ISR, all interrupts are disabled by the hardware. If the ISR for interrupt j re-enables interrupt i and interrupt i arrives during the execution of ISR j, the execution of ISR i must finish before the execution of ISR j can resume. The *nested interrupts* (nested-isr) relation is defined as follows: for all memory events e_1, e_2 such that tid(e_1) $\neq$ tid(e_2) and enables(e_2, e_1), if $(e_1, e_2) \in$ ws $\bigcup$ rf $\bigcup$ fr, then for all events e such that $(e_1, e) \in$ po, we have $(e, e_2) \in$ nested-isr. This means if e in interrupt handler call ISR_i globally happens before e' in a different interrupt handler call ISR_j that enables ISR_i (represented by an edge from e to e' in the event graph), then all memory events e'' in ISR_i after e in the program order po must also globally happen before e', thus $(e'', e') \in$ nested-isr.

Algorithm 1 computes the constraints of relation *write serialisation* (ws), which is a total order between write events that access the same memory in the set C_{ws}, and the corresponding part of the relation *write serialisation* (nested-isr)

Algorithm 1 Constraints for write serialisation

```
Input: A set of symbolic events and program order po
Output: Set of contraints $C_{ws}$, $C_{isr\_ws}$
 1. $C_{ws} := \emptyset$; $C_{isr\_ws} := \emptyset$
 2. $writes$ := $\{(\alpha, \{w_1 \ldots w_n\}) \mid w_i$ is a write event $\wedge$ addr$(w_i) = \alpha\}$
 3. foreach $\alpha$ s.t. $\exists W.(\alpha, W) \in writes$ do
 4.     foreach $w, w' \in W$ s.t. tid$(w) \neq$ tid$(w')$ do
 5.         $C_{ws} := C_{ws} \bigcup \{\neg c_{ww'} \Rightarrow c_{w'w}\}$
 6.         if enables$(w', w)$ then
 7.             $C_{isr\_ws} := C_{isr\_ws} \bigcup \{(c_{ww'} \wedge \mathrm{g}(w) \wedge \mathrm{g}(w'))$
                                 $\Rightarrow \bigwedge_{e \in last(w)} c_{ew'}\}$
 8.         if enables$(w, w')$ then
 9.             $C_{isr\_ws} := C_{isr\_ws} \bigcup \{(c_{w'w} \wedge \mathrm{g}(w') \wedge \mathrm{g}(w))$
                                 $\Rightarrow \bigwedge_{e \in last(w')} c_{ew}\}$
```

that is related to ws in C_{isr_ws}. Specifically, it constructs, for each write-write pair (w, w') that access the same memory location and do not belong to the same ISR call (lines 2–4), a constraint indicating that either one and only one of $(w, w') \in$ ws or $(w', w) \in$ ws can be true (line 5). Thus, a satisfying assignment of this constraint constitutes a total order on all write events that access α.

If a write w globally happens before another write w' ($c_{ww'}$ $\wedge$ g(w) $\wedge$ g(w')) and enables(w', w) is true, according to our definition of relation *nested interrupts* (nested-isr), all memory events e after w in the program order must also globally happen-before w'. This is encoded as $\bigwedge_{e \in last(w)} c_{ew'}$, where $last(e)$ is defined to be

$$\{e' \mid (e, e') \in \mathrm{po} \wedge \neg\exists e''.\mathrm{tid}(e') = \mathrm{tid}(e'') \wedge (e,' e'') \in \mathrm{po}\}$$

in lines 6–7. Note that in the encoding we state only that the last event e in the program order of the ISR that contains w happens before w'. This is an optimisation, as all memory events between w and e in program order will happen before w' by transitivity. Symmetrically, we also need a similar constraint in cases where enables(w, w') is true.

Algorithm 2 Constraints for read-from

```
Input: A set of symbolic events and program order po
Output: Sets of contraints $C_{rf}$, $C_{isr\_rf}$
 1. $C_{rf} := \emptyset$; $C_{isr\_rf} := \emptyset$
 2. $reads$ := $\{(\alpha, \{r_1 \ldots r_n\}) \mid r_i$ is a read event $\wedge$ addr$(r_i) = \alpha\}$
 3. $writes$ := $\{(\alpha, \{w_1 \ldots w_n\}) \mid w_i$ is a write event $\wedge$ addr$(w_i) = \alpha\}$
 4. foreach $\alpha$ s.t. $\exists R, W.(\alpha, R) \in reads \wedge (\alpha, W) \in writes$ do
 5.     foreach $r \in R$ do
 6.         $rf\_tmp := \emptyset$
 7.         foreach $w \in W$ do
 8.             if $(r, w) \notin$ po then
 9.                 $rf\_tmp := rf\_tmp \bigcup \{s_{wr}\}$
10.                 $C_{rf} := C_{rf} \bigcup \{s_{wr} \Rightarrow (\mathrm{g}(w) \wedge \mathrm{g}(r) \wedge \mathrm{val}(w) = \mathrm{val}(r))\}$
11.                 $C_{rf} := C_{rf} \bigcup \{s_{wr} \Rightarrow c_{wr}\}$
12.                 if tid$(w) \neq$ tid$(r) \wedge$ enables$(r, w)$
13.                     $C_{isr\_rf} := C_{isr\_rf} \bigcup \{s_{wr} \Rightarrow \bigwedge_{e \in last(w)} c_{er}\}$
14.         $C_{rf} := C_{rf} \bigcup \{\mathrm{g}(r) \Rightarrow \bigvee_{s \in rf\_tmp} s\}$
```

Algorithm 2 encodes the relation *read-from* (rf) as a set of constraints in C_{rf}, and the corresponding part of *nested interrupts* (nested-isr) in C_{isr_rf}. The algorithm, in addition to specifying possible orders between write and read events, relates the (symbolic) value of all read events to one of the possible writes. More precisely, for each write-read pair (w, r) that access the same memory location (lines 2–7), we use an auxiliary variable s_{wr} (line 9) to indicate that r reads the value from w (line 10) as well as say that w must globally happen before r (line 11). For each occurrence of a shared variable, we introduce a fresh index in the SSA encoding. Hence, reading a shared variable is non-deterministic. By stating that the symbolic values of r and w are equal in our encoding (i.e. val(r) = val(w) in line 10), we enforce the value read from any shared variable to be consistent with the corresponding write. The constraints related to nested interrupts are generated in the same way as Algorithm 1 (lines 12–13).

Algorithm 3 Constraints for from-read

```
Input: A set of symbolic events and program order po
Output: Sets of contraints $C_{fr}$, $C_{isr\_fr}$
 1. $C_{fr} := \emptyset$; $C_{isr\_fr} := \emptyset$
 2. $reads$ := $\{(\alpha, \{r_1 \ldots r_n\}) \mid r_i$ is a read event $\wedge$ addr$(r_i) = \alpha\}$
 3. $writes$ := $\{(\alpha, \{w_1 \ldots w_n\}) \mid w_i$ is a write event $\wedge$ addr$(w_i) = \alpha\}$
 4. foreach $\alpha$ s.t. $\exists R, W.(\alpha, R) \in reads \wedge (\alpha, W) \in writes$ do
 5.     foreach $(w', w) \in W \times W$, $r \in R$ s.t. $w \neq w' \wedge$ tid$(r) \neq$ tid$(w)$ do
 6.         $C_{fr} := C_{fr} \bigcup \{(s_{w'r} \wedge c_{w'w} \wedge \mathrm{g}(w) \wedge \mathrm{g}(w')) \Rightarrow c_{rw}\}$
 7.         if enables$(w, r)$ then
 8.             $C_{isr\_fr} := C_{isr\_fr} \bigcup \{(s_{w'r} \wedge c_{w'w} \wedge \mathrm{g}(w))$
                                 $\Rightarrow \bigwedge_{e \in last(r)} c_{ew}\}$
```

Finally, **Algorithm 3** constructs the constraints for the relation *from-read* (fr) in C_{fr} that specifies orders between read and write events, and the part of *nested interrupts* (nested-isr) related to fr in C_{isr_fr}: for every read-write pair (r, w) that accesses memory address α (lines 2–4) and for every write w' that accesses the same memory and globally happens before w, i.e., $(w', w) \in$ ws (encoded as $c_{w'w}$ $\wedge$ g(w) $\wedge$(w') in line 6), if r gets its value from w' (recall in Algorithm 2 we use the auxiliary variable $s_{w'r}$ to encode $(w', r) \in$ rf), then c_{rw} must hold, that is r must happen before w (otherwise it would get its value from w). The constraints related to nested interrupts are generated similarly to the algorithms described previously (lines 7–8).

Our encoding does not relate events in the same ISR call as they are already related by program order (except for *read-from* as a read event can get its value from a write event in the same ISR call). The (asymptotic) size of our encoding is, in the worst case, quadratic for *write serialisation* and *read-from*, and cubic for *from-read*, in the maximum number of memory events for a *single* address. Note that the encoding produced by our algorithm is by no means minimal in terms of size. For example, line 9 of Algorithm 2 can, in principle, exclude cases where all of the following hold: w and r do not belong to the same ISR call, enables(r, w) is true, and w is the last write event in the program order s.t. the conjunction of guards g(w) and g(r) is satisfiable. However, in each iteration, we would need to check whether g(w) $\wedge$ g(r) is satisfiable, which can be quite expensive. We need to balance the size of the encoding and the computational cost of producing it.

IV. Experiments

We compare our new partial-order encoding, described in the preceding section, with two conventional source-to-source transformations. These translate interrupt-driven code into sequential code and into multi-threaded code.

A. Interrupt Sequentialisation

The first conventional approach we compare our new method with is a source-to-source transformation that translates a program with interrupts into a sequential program, similar in style to the approach in [8]. The transformation is realised by insertion of calls to a scheduling function that models nested interrupt arrivals and handling. This allows us to leverage existing tools for verifying sequential programs. Partial-order reduction is applied to reduce the number of calls to the scheduling function. The scheduling function is inserted before and after a statement where a write dependency occurs, and is inserted before the statement if it is a read dependency [3].

We define a scheduling function with two global variables: `count` is the total number of interrupt arrivals during program execution, and `irq_enabled[i]` indicates whether interrupt `i` is enabled. At the beginning of the interrupt-driven code, `count` is set to a positive integer and array `irq_enabled` is initialised to zero. Then, `irq_enabled[i]` is set to 1 right after the statement in the original code that enables interrupt i. The scheduling function does the following. First, it checks whether `count` is zero. If it is, the scheduler does nothing. Otherwise, an interrupt might occur; to decide which one, the scheduler makes a non-deterministic choice among the interrupts whose bits in `irq_enabled` are set to 1. If there is such an interrupt, the scheduler makes a non-deterministic choice between doing nothing (to simulate that no interrupts occur) or calling the corresponding ISR, followed by decrementing `count`. In the special case where interrupt priorities are explicit, we use the variables `curr_pty` and `prev_pty` to record the current priority, and the previous priority upon return from the current ISR, respectively, in the same way as [8].

Although in our experiments we bound the number of interrupt arrivals, the sequential translation can be lifted to handle an unlimited number of interrupt arrivals by adding an infinite loop in the scheduling function.

B. Instrumentation with Threads

The second source-to-source transformation that we use for comparison generates multi-threaded code, and was suggested in [11]. The invocation of the ISR is modelled by spawning a thread that executes the ISR, which is instrumented as follows:

```
irqreturn_t handler_i(param_list) {
  atomic { irq_sleep[i] = 0; }
  /* original code */
  ...
  atomic { irq_sleep[i] = 1; }
  return IRQ_HANDLED;
}
```

Function calls are inlined and an atomic statement `atomic_assume(irq_sleep[0] && ... && irq_sleep[i-1])` is inserted between every single statement in the inlined code of the ISRs, assuming that if $i < j$ then interrupt handler call i is preempted by interrupt handler call j by enabling interrupt j in handler i or in other handler calls that preempt i. In the special case where interrupts have explicit priorities, we assume that interrupt i has a higher priority than interrupt j if $i < j$. We set `irq_sleep[i] = 0` if and only if the ISR call i is executing or preempted by other ISR calls. Different calls to the same ISR require separate function bodies with different instrumentations.

C. Verification Tools

To evaluate the performance of our partial-order encoding for nested interrupts, we implemented our algorithms in a prototype tool called *i-CBMC*, which is an extension of the CBMC Bounded Model Checker [5]. Moreover, we apply a range of existing verifiers to sequentialised programs and programs that have been instrumented with threads.

For *interrupt sequentialisation* as described in Section IV-A, we used the following tools: SatAbs [6], BLAST, CPAChecker, UFO and the sequential version of CBMC [5]. For interrupt *instrumentation with threads* as discussed in Section IV-B, we used: the concurrent version of SatAbs, Threader, IMPARA [13], ESBMC, and the concurrent version of CBMC [1]. The tools were run using the 2014 SV-COMP configuration.[1] Bounded model checkers were applied with one loop unwinding. We make our implementation, our benchmarks and detailed experimental data available for other researchers at http://www.cprover.org/interrupts/.

D. Benchmarks

We assess the effectiveness of each method on benchmarks derived from embedded software and Linux device drivers.

For each benchmark, we have a version where interrupts are correctly disabled and enabled, and another version where they are *incorrectly* managed, and hence, safety properties are violated.

Logger is code modelled after parts of the firmware of a temperature logging device from a major industrial enterprise (112 LOC, and 172 LOC for the extended version). It has two interrupt-triggered tasks: the measurement task should not be preempted by the communication task. Otherwise, a measurement is written to a wrong position in the log.

Blink is an example application from the TinyOS[2] distribution (2652 LOC). It displays a pattern on three LEDs with the help of timers. TinyOS provides a virtualisation API to handle several alarms with one hardware timer. There are two interrupt-triggered tasks, for firing the alarm and updating the timer when it overflows. If the alarm firing can preempt the updating of the timer, the timer keeps running even though the alarm has already fired.

Brake is code generated from the Simulink model of a brake-by-wire system provided by Volvo Technology AB (3938 LOC). The system consists of four threads communicating with four wheel brake controllers, together with one main thread, responsible for computing the braking torque, which should not be preempted. Otherwise, the brake torque can be miscalculated. We parameterised this benchmark by the number of wheels.

RcCore is a Linux device driver for a remote control together with a model of the Linux Kernel (7035 LOC). The original driver uses locks to enforce priorities—namely that querying the availability of transmission protocols may not be interrupted by a modification to the protocol settings. We introduced a bug such that the driver may crash due to a NULL pointer dereference if priorities are disregarded.

We performed all experiments on a 64-bit machine running Linux 3.15.7 with eight Intel Xeon 3.07 GHz cores and 48 GB of main memory.

[1] http://sv-comp.sosy-lab.org/2014/rules.php

[2] http://www.tinyos.net

			Sequentialisation				Instrumentation with Threads			PO
	Lines of Code	No. of Interrupts	BLAST 2.7.2	CPAChecker 1.3.4	UFO SV-COMP14	CBMC r4781	IMPARA r878	ESBMC 1.23	CBMC r4781	i-CBMC
Logger	112	2	✓ 17.0 s	✓ 1.8 s	✓ 1.4 s	TO	✓ **0.2 s**	✓ 2.5 s	✓ 0.4 s	✓ **0.2 s**
+ incorrect	112	2	! 26.7 s	? 2.0 s	! 36.2 s	TO	! 0.4 s	! **0.2 s**	! 1.1 s	! **0.2 s**
Logger (extended)	172	3	✗	✓ **2.1 s**	TO	TO	✓ 65.2 s	TO	✓ 250.6 s	✓ 25.2 s
+ incorrect	172	3	TO	? 2.4 s	TO	TO	! 117.1 s	TO	! 318.5 s	! **22.5 s**
Blink	2,652	2	✗	unknown	✓ 1425.1 s	TO	✓ 360.8 s	TO	✓ 4.5 s	✓ **3.6 s**
+ incorrect	2,652	2	✗	unknown	? 1420.5 s	TO	! 37.3 s	TO	? 14.6 s	! **4.2 s**
RcCore	7,035	3	* 41.1 s	unknown	TO	TO	✗	TO	✓ 87.3 s	✓ **75.7 s**
+ incorrect	7,035	3	! **37.1 s**	unknown	TO	✗	✗	✗	! 94.4 s	! 75.5 s
Brake (1 Wheel)	3,938	2	* 0.8 s	* 3.1 s	✓ **0.8 s**	✗	TO	✗	✓ 42.2 s	✓ 154.2 s
+ incorrect	3,938	2	TO	unknown	? 1.1 s	✗	TO	✗	! 7.2 s	! **3.7 s**
Brake (2 Wheels)	3,938	3	TO	* 5.6 s	✓ **7.8 s**	✗	TO	✗	✓ 797.3 s	TO
+ incorrect	3,938	3	TO	unknown	? 1.1 s	✗	TO	✗	! 29.3 s	! **6.7 s**
Brake (3 Wheels)	3,938	4	TO	* 10.6 s	✓ **854.1 s**	✗	TO	TO	TO	TO
+ incorrect	3,938	4	TO	unknown	? 3.8 s	✗	TO	TO	! 54.8 s	! **9.8 s**

✓= proved correct, ! = bug exposed, ? = bug missed, * = false alarm, ✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

TABLE I: Experimental results of three approaches for verifiying programs with nested interrupts

E. Results and Discussion

Table I presents the results of our performance comparison. Column 1 lists our five benchmarks: Logger, extended Logger, Blink, RcCore and Brake (parametrised by the number of wheels). Columns 2 and 3 give the number of lines of code and the number of interrupt arrivals we modelled in each benchmark. Column 4 gives results for the interrupt sequentialisation (Section IV-A) and Column 5 gives the results for the instrumentation with threads (Section IV-B). Finally, Column 6 gives the results of our new partial-order encoding (Section III) implemented in the prototype tool i-CBMC. SatAbs crashed and Threader reported parse errors on all our benchmarks so we did not include them in the table. The best correct entry for each benchmark is marked in bold.

Our results indicate that the transformation to sequential code performs worst, whereas our partial order encoding is clearly more effective than the two program transformation techniques.

In our experiments with *interrupt sequentialisation*, the number of interrupt arrivals is bounded. Only BLAST and UFO are able to handle both versions of the simplest benchmark Logger correctly. For the other benchmarks, the tools we tried were able to obtain only partial results. It is worth noting that not all the tools can handle the recursion in the scheduling function. CBMC was not able to finish unwinding the recursion after 30 minutes. The reason is that non-determinism is encoded directly into a SAT formula. Recursive function calls were skipped by CPAChecker.

The results of *interrupt instrumentation with threads* are better compared to sequentialisation: ESBMC can handle only the simplest benchmark Logger. IMPARA was able to return the expected results of the two Logger and the Blink benchmarks. CBMC managed to report the expected results in most versions of our benchmarks. It failed to return a safety violation of the faulty Blink.

A benefit of using instrumentation with threads is that it can model concurrent execution of interrupts and threads, whereas interrupt sequentialisation cannot. However, different calls to the same interrupt handler normally require separate ISR function bodies with different instrumentations.

Overall, *i-CBMC with partial-order encoding* is the winner, outperforming other approaches. It times out in the non-faulty Brake benchmark with two and three wheels, and yields the expected results in all other experiments. For benchmarks on which other tools returned the correct result with less runtime than i-CBMC, none of them were able to return the correct results in both the non-faulty and faulty versions of the same benchmark.

In our instrumentation with threads, we made each line of code atomic, which is an under-approximation of the behaviours of the original code. Our partial-order encoding models reads and writes as different atomic events and hence captures more behaviours. Yet, the runtime of i-CBMC is significantly better than that of CBMC with instrumentation for most variants of our benchmarks. The reason is that the assume statements in the multi-threaded instrumentation increase the size of the SAT formula that CBMC generates substantially.

Just like the interrupt instrumentation with threads, i-CBMC supports concurrent executions of interrupts (e.g., interrupts on different CPU cores) and threads by simply dropping the constraints for nested interrupts between interrupts (threads) i and j if their executions can interleave. Another advantage is that for different invocations of the same ISR, i-CBMC does not require separate function bodies (whereas the thread instrumentation does). This case is simply modelled using a function call to the ISR. A limitation of the partial-order encoding is that it is based on bounded model checking, and thus is able to verify only a bounded number of interrupts and threads. By contrast, the sequentialised programs can be passed to unbounded model checkers.

As the partial-order encoding relates only memory events that access the same shared memory location, the counterexamples i-CBMC generates may exhibit context switches from an interrupt handler to another one that enables it. However, this is only a matter of presentation of the counterexample trace: the statements that manipulate local variables or variables that access different memory locations are unordered w.r.t. different interrupt handlers. Since i-CBMC displays a linearised counterexample trace, it has to choose a total ordering. Thus, they may appear interleaved, although they are actually unordered.

V. Related Work

Previous research on formal verification of interrupt-driven programs uses a diverse range of techniques, including program transformation [8], [11], [15], explicit-state model checking [12], bounded model checking [3], [9] and predicate abstraction [14]. None of them can effectively verify programs with nested interrupts of moderate size. We briefly survey those methods closely related to our techniques.

Wu et al. [15] describe a translation from programs with nested interrupts into sequential code, which makes the conservative assumption that interrupts may arrive after every instruction. This approach suffers from the state explosive problem. Kidd et al. [8] introduce a program transformation that translates a multi-threaded program into a sequential one by encoding a thread scheduler. Their method covers threads with fixed priorities, and so is also applicable to interrupt-driven programs with fixed interrupt priorities. Sec. IV-A discusses a modified version of their encoding, in which the number of interrupt arrivals is a parameter and multiple arrivals of the same interrupt are allowed. We also provide experimental results for such a transformation. By contrast, [8] employs the notion of a “hyperperiod”, during which every interrupt happens exactly once.

Regehr [11] suggests a source-to-source transformation that turns interrupt-driven code into multi-threaded code by assigning a fixed priority to a thread when it is created. During program execution, threads are scheduled according to their predefined priorities. This approach, however, requires a thread-aware verification tool that can model threads with static priorities. We provide experimental results for a variant of their approach in which the thread priorities are enforced by adding further instrumentation.

Burcur and Kwiatkowska [3] present a platform-dependent, OS-independent verification tool for programs written in C with interrupts. Similar to Schlich et al. [12] and Li et al. [9], they use partial order reduction to reduce the number of interrupt handler calls. However, nested interrupts are not supported. The instrumented program is checked using CBMC. Based on the semantics given in [2], Alglave et al. [1] introduce a symbolic encoding of concurrent programs that supports not only SC but also weaker memory models such as Intel’s x86 and IBM’s Power. By contrast, the encoding proposed in this paper is a *strengthening* of SC.

VI. Conclusion

We propose a novel method to verify programs with nested interrupts in the form of symbolic execution based on a partial-order encoding that precisely models the interleaving semantics of nested interrupts. We experimentally compare the proposed method to conventional approaches based on source-to-source transformations to sequential or multi-threaded programs that can be handled by off-the-shelf verification tools. Our experimental results demonstrate that our partial-order encoding is the most effective method to verify software with nested interrupts.

For future work, we will implement a field-sensitive SSA encoding in i-CBMC, which has the potential to greatly reduce the number of shared variables in our partial-order encoding, and will therefore improve scalability. We used finite loop unwindings in our experiments. We plan to implement our method in the model checker Impara [13], which is capable of proving unbounded safety.

Acknowledgements: This work is funded by a gift from Intel Corporation for research on *Effective Validation of Firmware* and the ARTEMIS VeTeSS project. We are grateful for illuminating discussions with Luke Ong (Oxford), Alan Hu (UBC), Moshe Vardi (Rice) and Sharad Malik (Princeton). We thank Ilja Zakharov (ISPRAS) and Doina Bucur (Groningen) for prividing us the initial version of the RcCore and Blink benchmarks, respectively, and Vincent Nimal (Oxford) for fruitful discussions on CBMC.

References

[1] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient Bounded Model Checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157, 2013.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models (extended version). *FMSD*, 40(2):170–205, 2012.

[3] D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Software and Systems*, 84(10):1693–1707, 2011.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.

[6] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, pages 570–574, 2005.

[7] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.

[8] N. Kidd, S. Jagannathan, and J. Vitek. One stack to run them all: Reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN*, LNCS, pages 245–261. Springer, 2010.

[9] C. Li, A. Raghunathan, and N. K. Jha. Improving the trustworthiness of medical device software with formal verification methods. In *Embedded Systems Letters*, pages 50–53, 2013.

[10] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *ASPLOS*, pages 305–318, 2011.

[11] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *ENTCS*, 174(9):139–150, 2007.

[12] B. Schlich, T. Noll, J. Brauer, and L. Brutschy. Reduction of interrupt handler executions for model checking embedded software. In *HVC*, pages 5–20. Springer, 2009.

[13] B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with Impact. In *FMCAD*, pages 210–217, 2013.

[14] T. Witkowski, N. Blanc, G. Weissenbacher, and D. Kroening. Model checking concurrent Linux device drivers. In *ASE*, pages 501–504. IEEE, 2007.

[15] X. Wu, Y. Wen, L. Chen, W. Dong, and J. Wang. Data race detection for interrupt-driven programs via bounded model checking. In *Software Security and Reliability-Companion*, pages 204–210, 2013.