Modular Verification of Interrupt-Driven Software

Chungha Sung
University of Southern California
Los Angeles, CA, USA

Markus Kusano
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

***Abstract*—Interrupts have been widely used in safety-critical computer systems to handle outside stimuli and interact with the hardware, but reasoning about interrupt-driven software remains a difficult task. Although a number of static verification techniques have been proposed for interrupt-driven software, they often rely on constructing a monolithic verification model. Furthermore, they do not precisely capture the complete execution semantics of interrupts such as nested invocations of interrupt handlers. To overcome these limitations, we propose an *abstract interpretation* framework for static verification of interrupt-driven software that first analyzes each interrupt handler in isolation as if it were a sequential program, and then propagates the result to other interrupt handlers. This *iterative* process continues until results from all interrupt handlers reach a fixed point. Since our method never constructs the global model, it avoids the up-front blowup in model construction that hampers existing, non-modular, verification techniques. We have evaluated our method on 35 interrupt-driven applications with a total of 22,541 lines of code. Our results show the method is able to quickly and more accurately analyze the behavior of interrupts.**

I. Introduction

Interrupts have been widely used in safety-critical embedded computing systems, information processing systems, and mobile systems to interact with hardware and respond to outside stimuli in a timely manner. However, since interrupts may arrive non-deterministically at any moment to preempt the normal computation, they are difficult for developers to reason about. The situation is further exacerbated by the fact that interrupts often have different priority levels: high-priority interrupts may preempt low-priority interrupts but not vice versa, and interrupt handlers may be executed in a nested fashion. Overall, methods and tools for accurately modeling the semantics of interrupt-driven software are still lacking.

Broadly speaking, existing techniques for analyzing interrupts fall into two categories. The first category consists of techniques based on testing [12], [30], which rely on executing the program under various interrupt invocation sequences. Since it is often practically infeasible to cover all combinations of interrupt invocations, testing will miss important bugs. The second category consists of static verification techniques such as model checking [3], [18], [33], [36], [43], which rely on constructing and analyzing a formal model. During the modeling process, interrupt-related behaviors such as preemption are considered. Unfortunately, existing tools such as iCBMC [18] need to bound the execution depth to remain efficient, which means shallow bugs can be detected quickly, but these tools cannot prove the absence of bugs.

In this paper, we propose a static verification tool geared toward proving the absence of bugs based on *abstract interpretation* [5]. The main advantage of abstract interpretation is the sound approximation of complex constructs such as loops, recursions and numerical computations. However, although abstract interpretation techniques have been successfully applied to sequential [6] and multithreaded software [27], they have not been able to precisely model the semantics of interrupt-driven software.

At the high level, interrupts share many similarities with threads, e.g., both interrupt handlers and thread routines may be regarded as sequential programs communicating with others via the shared memory. However, there are major differences in the way they interleave. For example, in most of the existing verification tools, threads are allowed to freely preempt each other's execution. In contrast, interrupts often have various levels of priority: high-priority interrupts can preempt low-priority interrupts but not vice versa. Furthermore, interrupts with the same level of priority cannot preempt each other. Thus, the behavior manifested by interrupts has to be viewed as a subset of the behavior manifested by threads.

To accurately analyze the behavior of interrupts, we develop IntAbs, an *iterative* abstract interpretation framework for interrupt-driven software. That is, the framework always analyzes each interrupt handler in isolation before propagating the result to other interrupt handlers and the *per interrupt* analysis is iterated until results on all interrupt handlers stabilize, i.e., they reach a *fixed point*. Thus, in contrast to traditional techniques, it never constructs the monolithic verification model that often causes exponential blowup up front. Due to this reason, our method is practically more efficient than these traditional verification techniques.

The IntAbs framework also differs from prior techniques for statically analyzing interrupt-driven software, such as the source-to-source transformation-based testing approach proposed by Regehr [31], the sequentialization approach used by Wu et al. [42], and the model checking technique implemented in iCBMC [18]. For example, none of these existing techniques can soundly handle infinite loops, nested invocations of interrupts, or prove the absence of bugs. Although some prior abstract interpretation techniques [28] over-approximate of the interrupt behavior, they are either non-modular or too inaccurate, e.g., by allowing too many infeasible *store-to-load* data flows between interrupts. In contrast, our approach precisely models the preemptive scheduling of interrupts to identify apparently-infeasible data flows. As shown in Fig. 1, by pruning away these infeasible data flows, we can drastically improve the accuracy of the overall analysis.

IntAbs provides not only a more accurate modeling of the interrupt semantics but also a more efficient abstract interpretation framework. We have implemented IntAbs in a static

978-1-5386-2684-9/17$15.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

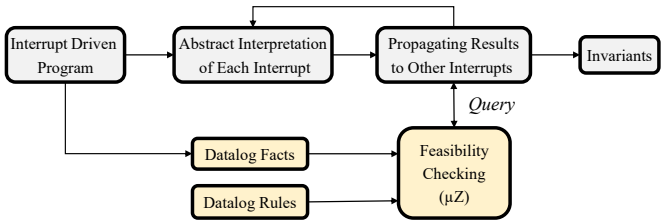


Fig. 1. IntAbs – iterative verification framework for interrupt-driven programs.

analysis tool for C/C++ programs, which uses Clang/LLVM [1] as the front-end, Apron [16] for implementing the numerical abstract domains, and μZ [13] for checking the feasibility of data flows between interrupts. We evaluated IntAbs on 35 interrupt-driven applications with a total of 22,541 lines of C code. Our experimental results show that IntAbs can efficiently as well as more accurately analyze the behavior of interrupts by removing a large number of infeasible data flows between interrupts.

In summary, the main contributions of our work are:

- A new abstract interpretation framework for conducting static verification of interrupt-driven programs.
- A method for soundly and efficiently identifying and pruning infeasible data flows between interrupts.
- The implementation and experimental evaluation on a large number of benchmark programs to demonstrate the effectiveness of the proposed techniques.

The remainder of this paper is organized as follows. We first motivate our approach in Section II by comparing it with testing, model checking, and abstract interpretation tools designed for threads. Then, we provide the technical background on interrupt modeling and abstract interpretation in Section III. Next, we present our new method for checking the feasibility of data flows between interrupts in Section IV, followed by our method for integrating the feasibility checking with abstract interpretation in Section V. We present our experimental evaluation in Section VI. Finally, we review the related work in Section VII and conclude in Section VIII.

II. Motivation

We first use examples to illustrate the problems of prior techniques such as testing, model checking, and thread-modular abstract interpretation. Then, we explain how our method overcomes these problems.

Consider the example program in Fig. 2, which has three interrupts `irq_H`, `irq_L` and `irq_M`. The suffix `H` is used to denote high priority, `L` for low priority, and `M` for medium priority. Interrupts with higher priority levels may preempt interrupts with lower priority levels, but not vice versa. Inside the handlers, there are two variables `x` and `y`, which are set to `0` initially. Among the three assertions, the first two may fail, while the last one always holds.

A. Testing

Testing an interrupt-driven program requires the existence of interrupt sequences, which must be generated a priori. In

```
irq_H() {
  ...
  assert(y==0);
}
```

```
irq_L() {
  x = 0;
  assert(x==0);
}
```

```
irq_M() {
  y = 1;
  x = 1;
  assert(x==1);
}
```

Fig. 2. An example program with three interrupt handlers and assertions.

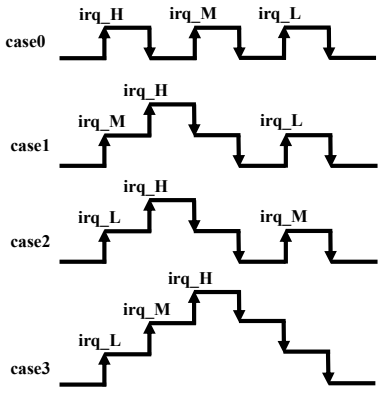


Fig. 3. Some possible interrupt sequences for the program in Fig. 2.

Fig. 2, for example, since the interrupt handlers have different priority levels, we need to consider preemption while creating the test sequences. Since a high-priority interrupt handler may preempt, at any time, the execution of a medium- or low-priority interrupt handler, when `irq_L` is executing, `irq_H` may be interleaved in between its instructions.

Fig. 3 shows four of the possible interrupt sequences for the program. Specifically, *case0* is the sequential execution of the three handler functions; *case1* shows that `irq_H` preempts `irq_M`, followed by `irq_L`; *case2* is similar, except that `irq_L` executes first and then is preempted by `irq_H`, followed by `irq_M`; and *case3* is the nested case where `irq_L` is preempted by `irq_M` and then by `irq_H`.

The main problem of testing is that there can be too many such interrupt sequences to explore. Even if we can somehow guarantee that each interrupt handler is executed only once, the total number of test sequences can be enormously large, even for small or medium-sized programs.

B. Model Checking

Model checking tools such as CBMC [4] may be used to search for erroneous interrupt sequences, e.g., those leading to assertion violations. For instance, in the running example, all assertions hold under the sequences *case0* and *case2* in Fig. 3. This is because, although `irq_H` preempts `irq_L`, they access different variables and thus do not affect the assertion conditions, while `irq_M` checks the value of `x` after assigning 1 to `x`.

In *case1*, however, the execution order of the three interrupt handlers is different, thus leading to an assertion violation inside `irq_H`. More specifically, `irq_M` is preempted by `irq_H` at first. Then, after both completes, `irq_L` is executed. So, the

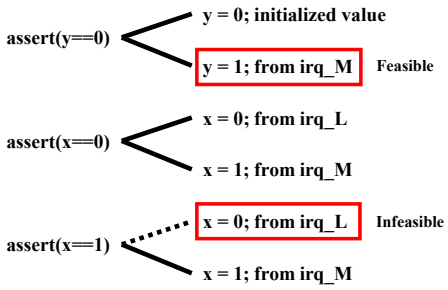


Fig. 4. Some possible *store-to-load* data flows during abstract interpretation.

change of `y` may affect the read of `y` in `irq_H`, leading to the violation.

Finally, in *case3*, both of the first two assertions may be violated, because the check of `x` in `irq_L` and the check of `y` in `irq_H` can be affected by `irq_M`'s own assignments of `x` and `y`.

Although bounded model checking can quickly find bugs, e.g., the assertion violations in Fig. 2, the depth of the execution is often bounded, which means in practice, tools such as iCBMC [18] cannot prove the absence of bugs.

C. Abstract Interpretation

Abstract interpretation is a technique designed for proving properties, e.g., assertions always hold. Unfortunately, existing methods based on abstract interpretation are mostly designed for threads as opposed to interrupts. Since threads interact with each other more freely than interrupts, these methods are essentially over-approximated analysis. As such, they may still be leveraged to prove properties in interrupt-driven programs, albeit in a less accurate fashion. That is, when they prove an assertion holds, the assertion indeed holds; but when they cannot prove an assertion, the result is inconclusive.

For the running example in Fig. 2, for instance, existing abstract interpretation techniques such as Miné [24], [27], designed for analyzing threads, cannot prove any of the three assertions. To see why, let us first assume that interrupt handlers are thread routines. During thread-modular abstract interpretation, the verification procedure would first gather all possible pairs of load and store instructions with respect to the global variables, as shown in Fig. 4, where each assertion has two possible loads. Specifically, the load of `y` in `irq_H` corresponds to the initial value `0` and the store in `irq_M`. The load of `x` in `irq_L` corresponds to the stores in `irq_L` and `irq_M`. The load of `x` in `irq_M` corresponds to the stores in `irq_L` and `irq_M`.

Since these existing methods [24], [27] assume that all stores may affect all loads, they would incorrectly report that all three assertions may fail. For example, it reports that the load of `x` in `irq_M` may (incorrectly) read from the store `x=0` in `irq_L` despite that `irq_L` has a lower priority and thus cannot preempt `irq_M`. In contrast, our new method can successfully prove the third assertion. Specifically, we model the behavior of interrupts with different levels of priorities.

```
irq_M() {          irq_L() {          irq_H() {
 if (...)           ...                if (...)
   y = 0;                                x = 0;
                    y = 1;
 y = 1;             ...                x = 1;

 assert(x==1);      assert(y==1);      assert(y==1);
}                  }                  }
```

Fig. 5. An example program with three interrupt handlers, where the first two assertions alway hold but the last assertion may fail.

Due to the different priority levels, certain *store-to-load* data flows are no longer feasible, as shown by the stores marked by red boxes in Fig. 4: these two stores have lower priority than the corresponding load in the assertions.

D. Abstract Interpretation for Interrupts

Modeling the priority levels *alone*, however, is not enough for proving all assertions because even without preemption, a low-priority interrupt may affect a high-priority interrupt. Consider the first red box in Fig. 4. Although `y=1` from `irq_M` cannot affect the load of `y` in `irq_H` through preemption, if `irq_H` is invoked after `irq_M` ends, `y` can still get the value `1`, thus leading to the assertion violation. Therefore, our new verification procedure has to consider all possible sequential interleavings of the interrupt handlers as well.

Now, consider the program in Fig. 5, which has three interrupt handlers `irq_M`, `irq_L` and `irq_H`. In these handler functions, there are two global variables `x` and `y`, which are set to `0` initially. Among the three assertions, the first two always hold, whereas the last one may fail. For ease of comprehension, we assume the computer hardware running this program provides the *sequentially consistent* memory [20], [22], [45]. Note that `irq_M` has two stores of `y`, one inside the conditional branch and the other outside, and `irq_H` has two stores of `x`, one inside the conditional branch and the other outside.

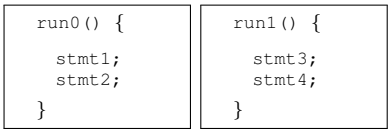
With prior thread-modular analysis [19], [24], [27], all three assertions may fail because the store `y=0` in `irq_M` may be interleaved right before the assertions in `irq_L` and `irq_H`. Furthermore, the store `x=0` in `irq_H` executed before `irq_M` may lead to the violation of the assertion in `irq_M`. In contrast, with our precise modeling of the interrupt behavior, the new method can prove that the first two assertions always hold. Specifically, the assertion in `irq_L` holds because, even if it is preempted by `irq_M`, the value of `y` remains `1`. Similarly, the assertion in `irq_M` holds because, even if it is preempted by `irq_H`, the store `x=1` post-dominates the store `x=0`, meaning the value of `x` remains `1` after `irq_H` returns.

In contrast, the assertion in `irq_H` may fail if `irq_H` preempts `irq_M` right after the conditional branch that sets `y` to `0`. This particular preemption is feasible because `irq_H` has a higher priority than `irq_M`.

Therefore, our new method has to consider not only the different levels of priority of all interrupts, but also the domination and post-domination relations within each handler. It decides the feasibility of *store-to-load* data flows based on whether a load has a dominated store, whether a store has a

TABLE I
COMPARING IntAbs WITH TESTING AND PRIOR VERIFICATION METHODS ON THE PROGRAMS IN FIG. 2 AND FIG. 5.

Property	Testing [12], [30]	Model Checking (bounded) [18], [42]	Abs. Int. for Threads [19], [27]	IntAbs for Interrupts (new)
assertion in Fig. 2: `irq_H`	violation	violation	warning	warning
assertion in Fig. 2: `irq_L`	violation	violation	warning	warning
assertion in Fig. 2: `irq_M`			(bogus) warning	**proof**
assertion in Fig. 5: `irq_M`			(bogus) warning	**proof**
assertion in Fig. 5: `irq_L`			(bogus) warning	**proof**
assertion in Fig. 5: `irq_H`	violation	violation	warning	warning



- Possible traces for interrupts:
  - `stmt1` → `stmt2` → `stmt3` → `stmt4`
  - `stmt1` → `stmt3` → `stmt4` → `stmt2`
- Possible traces for threads:
  - `stmt1` → `stmt2` → `stmt3` → `stmt4`
  - `stmt1` → `stmt3` → `stmt4` → `stmt2`
  - `stmt1` → `stmt3` → `stmt2` → `stmt4`

Fig. 6. The interleavings (after `stmt1`) allowed by interrupts and threads.

post-dominated store, and whether a load-store pair is allowed by the priority levels of the interrupts. We present the details of this *feasibility-checking* algorithm in Section IV.

To sum up, the main advantages of IntAbs over state-of-the-art techniques are shown in Table I. Specifically, testing and (bounded) model checking tools are good at detecting bugs (e.g., assertion violations) but cannot prove the absence of bugs, whereas thread-modular abstract interpretation tools are good at obtaining proofs, but may report many false positives (i.e., bogus warnings). In contrast, our new *abstract interpretation* method is significantly more accurate. It can obtain more proofs than prior techniques and, at the same time, can significantly reduce the number of bogus warnings.

III. PRELIMINARIES

In this section, we describe how interrupt-driven programs are modeled in our framework by comparing their behavior to the behavior of threads. Then, we review the basics of prior abstract interpretation techniques.

A. Modeling of Interrupts

We consider an interrupt-driven program as a finite set $T = \{T_1, \ldots, T_n\}$ of sequential programs. Each sequential program T_i, where $1 \leq i \leq n$, denotes an interrupt handler. For ease of presentation, we do not distinguish between the main program and the interrupt handlers. Globally, sequential programs in T are executed in a strictly interleaved fashion. Each sequential program may access its own local variables; in addition, it may access a set of global variables, through which it communicates with the other sequential programs in T.

The interleaving behavior of interrupts is a strict subset of the interleaving behavior of threads (c.f. [18]). This is because concurrently running threads are allowed to freely preempt each other's executions. However, this is not the case for interrupts.

Consider the example program in Fig. 6, which has two functions named `run0` and `run1`. If they were interrupts, where `run1` has a higher priority level than `run0`, then after executing `stmt1`, there can only be two possible traces. The first one is for `run1` to wait until `run0` ends, and the second one is for `run1` to preempt `run0`. If they were threads, however, there can be three possible traces after executing `stmt1`. In addition to the traces allowed by interrupts, we can also execute `stmt3` in `run1`, then execute `stmt2` in `run0`, and finally execute `stmt4` in `run1`. The third trace is infeasible for interrupts because the high-priority `run1` cannot be preempted by `stmt2` of the low-priority `run0`.

Since the interleaving behavior of interrupts is a strict subset of the interleaving behavior of threads, it is always safe to apply a sound static verification procedure designed for threads to interrupts. If the verifier can prove the absence of bugs by treating interrupts as threads, then the proof is guaranteed to be valid for interrupts. The result of this discussion can be summarized as follows:

Theorem 1: Since the interleaving behavior of interrupts is a subset of the interleaving behavior of threads, proofs obtained by any sound abstract interpretation over threads remain valid for interrupts.

However, the reverse is not true: a bug reported by the verifier that treats interrupts as threads may not be a real bug since the erroneous interleaving may be infeasible. In practice, there are also tricky corner cases during the interaction of interrupts, such as nested invocations of handlers, which call for a more accurate modeling framework for interrupts.

Interrupts may also be invoked in a nested fashion as shown by *case3* in Fig. 3, which complicates the static analysis. Here, we say interrupts are nested when one's handler function is invoked before another's handler function returns, and the third handler function is invoked before the second handler function returns. Such nested invocations are possible, for example, if the corresponding interrupts have different priority levels, where the inner most interrupt has the highest priority level. This behavior is different from thread interleaving; with numerous corner cases, it requires the development of dedicated modeling and analysis techniques.

B. Abstract Interpretation for Threads

Existing methods for modular abstract interpretation are designed almost exclusively for multithreaded programs [19], [24], [25], [27]. Typically, the analyzer works on each thread in isolation, without creating a monolithic verification model as in the non-modular techniques [7], [42], to avoid the up-front complexity blowup. At a high level, the analyzer iterates

Algorithm 1 Local analysis of T with prior interferences I.

```
1: function AnalyzeLocal(T = ⟨N, n0, δ⟩, I)
2:   S ← ∅                                   ▷ Map from nodes to states
3:   W ← {n0}                                ▷ Set of nodes to process
4:   while ∃n ∈ W do
5:     W ← W \ {n}
6:     if n is a shared-memory read of variable v then
7:       s ← tfunc(n, S(n) ⊔ I(v))
8:     else
9:       s ← tfunc(n, S(n))
10:    for all ⟨n, n'⟩ ∈ δ such that s ⋢ S(n') do
11:      S(n') ← S(n') ⊔ s
12:      W ← W ∪ {n'}
13:  return S
```

Algorithm 2 Analysis of the entire program, i.e., a set of T's.

```
1: function AnalyzeProg(P)
2:   S ← map all nodes to ⊥
3:   S' ← S
4:   repeat
5:     S = S'
6:     for all T ∈ P do
7:       I ← ⊎ Interf(T', S) for each T' ∈ P, T' ≠ T
8:       S' ← S' ⊎ AnalyzeLocal(T, I)
9:   until S' = S
10: function Interf(T = ⟨N, n0, δ⟩, S)
11:   I ← ∅
12:   for all n ∈ N do
13:     if n is a shared memory write to variable v then
14:       I(v) ← I(v) ⊔ tfunc(n, S(n))
15:   return I
```

through threads in two steps: (1) analyzing each thread in isolation, and (2) propagating results from the shared-memory writes of one thread to the corresponding reads of other threads.

Let the entire program P be a finite set of threads, where each thread T is represented by a control-flow graph $\langle N, n_0, \delta \rangle$ with a set of nodes N, an entry node n_0, and the transition relation δ. Each pair $(n, n') \in \delta$ means control may flow from n to n'. Each node n is associated with an abstract memory-state over-approximating the possible concrete states at n. We assume the abstract domain (e.g., intervals) is defined as a lattice with appropriate top ($\top$) and bottom ($\bot$) elements, a partial-order relation ($\sqsubseteq$), and widening/narrowing operators to ensure that the analysis eventually terminates [5]. We also define an interference I that maps a variable v to the values stored into v by some thread T.

Algorithm 1 shows how a thread-local analyzer works on T assuming some interferences I provided by the environment (e.g., writes in other threads). It treats T as a sequential program. Let $S(n)$ be the abstract memory-state at node n, n_0 be the entry node of T, and W be the set of nodes in T left to be processed. The procedure keeps removing node n from the work-list W and processing it until W is empty (i.e., a fixed point is reached).

If node n corresponds to a shared-memory read of variable v, then the transfer function TFUNC (Line 7) assumes that n can read either the local value (from S) or the value written by another thread (the interference $I(v)$). The transfer function TFUNC of an instruction n takes some memory-state as input and returns a new memory-state as output; the new memory-state is the result of executing the instruction in the given memory-state. Otherwise, if n is a local read, in which case the transfer function TFUNC uses the local memory-state (Line 9) as in the abstract interpretation of any sequential program. The analysis result (denoted S) is an over-approximation of the memory states within T assuming interferences I.

The procedure that analyzes the entire program is shown in Algorithm 2. It first analyzes each thread, computes the interferences, and then analyzes each thread again in the presence of these interferences. The iterative process continues until a fixed point on the memory-states of all threads is reached. Initially, S maps each node in the program to an empty memory-state $\bot$. S' contains the analysis results after one iteration of the fixed-point computation. The function INTERF returns the interferences of thread T, i.e., a map from some variable v to all the (abstract) values stored into v by T. Each thread T is analyzed in isolation by the loop at Lines 4–9. Here, we use $\uplus$ to denote the join ($\sqcup$) of all memory-states on the matching nodes.

This *thread-modular* abstract interpretation framework, while more efficient than monolithic verification, is potentially less accurate. For example, a load l may see *any* value written into the shared memory by a store s even if there does not exist a path in the program where l observes s. This is why, as shown in Table I, techniques such as [19], [27] cannot obtain proofs for the programs in Fig. 2 and Fig. 5. In the context of interrupt handlers with priorities, it means that even infeasible store-to-load flows due to priorities may be included in the analysis, thus causing false alarms.

In the remainder of this paper, we show how to introduce priorities into the propagation of data flows between interrupts during the analysis, thereby increasing the accuracy while retaining its efficiency.

IV. Feasibility of Data Flows between Interrupts

In this section, we present our method for precisely modeling the priority-based interleaving semantics of interrupts, and deciding the feasibility of *store-to-load* data flows between interrupts. If, for example, a certain *store-to-load* data flow is indeed not feasible, it will not be propagated across interrupts in Algorithm 2.

More formally, given a set of *store-to-load* pairs, we want to compute a new MUSTNOTREADFROM relation, such that MUSTNOTREADFROM(l, s), for any load l and store s, means if we respect all the other existing *store-to-load* pairs, then it would be infeasible for l to get the value written by s.

We have developed a Datalog-based declarative program analysis procedure for computing MUSTNOTREADFROM. Toward this end, we first generate a set of *Datalog facts* from the program and the given *store-to-load* pairs. Then, we generate a set of *Datalog rules*, which infer the new MUSTNOTREADFROM relation from the Datalog facts. Finally, we feed the facts together with the rules to an off-the-shelf Datalog engine, which computes the MUSTNOTREADFROM relation. In our implementation, we used the μ-Z Datalog engine [13] to solve the Datalog constraints.

A. Inference Rules

Before presenting the rules, we define some relations:

- $\text{DOM}(a, b)$: statement a dominates b in the CFG of an interrupt handler function.
- $\text{POSTDOM}(a, b)$: statement a post-dominates b in the CFG of an interrupt handler function.
- $\text{PRI}(s, p)$: statement s has the priority level p.
- $\text{LOAD}(l, v)$: l is a load of global variable v.
- $\text{STORE}(s, v)$: s is a store to global variable v.

Dominance and post-dominance are efficiently computable [8] within each interrupt handler (not across interrupt handlers). Priority information for each interrupt handler, and thus all its statements, may be obtained directly from the program. Similarly, LOAD and STORE relations may be directly obtained from the program.

Next, we present the rules for inferring three new relations: NOPREEMPT, COVEREDLOAD and INTERCEPTEDSTORE.

a) NOPREEMPT*:* The relation means s_1 cannot preempt s_2, where s_1 and s_2 are instructions in separate interrupt handlers. From the interleaving semantics of interrupts, we know a handler may only be preempted by another handler with a higher priority. Thus,

$$\text{NOPREEMPT}(s_1, s_2) \leftarrow \text{PRI}(s_1, p_1) \wedge \text{PRI}(s_2, p_2) \wedge (p_2 \geq p_1)$$

Here, $\text{PRI}(s_1, p_1)$ means s_1 belongs to a handler with priority p_1, and $\text{PRI}(s_2, p_2)$ means s_2 belongs to a handler with priority p_2. If p_1 is not higher than p_2, then s_1 cannot preempt s_2.

b) COVEREDLOAD*:* The relation means a load l of a variable v is covered by a store s to v inside the same interrupt handler; this is the case when s occurs before l along all program paths. This is captured by the *dominance* relation in the corresponding control flow graph:

$$\text{COVEREDLOAD}(l) \leftarrow \text{LOAD}(l, v) \wedge \text{STORE}(s, v) \wedge \text{DOM}(s, l)$$

c) INTERCEPTEDSTORE*:* The relation is similar to COVEREDLOAD. We say a store s_1 is intercepted by another store s_2 if s_2 occurs after s_1 along all program paths in the same handler. Intuitively, the value written by s_1 is always overwritten by s_2 before the handler terminates. Formally,

$$\text{INTERCEPTEDSTORE}(s_1) \leftarrow \text{STORE}(s_1, v) \wedge \text{STORE}(s_2, v) \wedge \text{POSTDOM}(s_2, s_1)$$

Finally, the MUSTNOTREADFROM relation is deduced using all aforementioned relations including NOPREEMPT, COVEREDLOAD and INTERCEPTEDSTORE. It indicates that, under the current situation (defined by the set of existing store-to-load data flows), a load l cannot read from a store s in any feasible interleaving. There are several cases:

First, we say a load l covered by a store in a handler I cannot read from a store s intercepted by another store in a handler I', because l cannot read from s using any preemption or by running I and I' sequentially.

$$\text{MUSTNOTREADFROM}(l, s) \leftarrow \text{COVEREDLOAD}(l) \wedge \text{LOAD}(l, v) \wedge \text{STORE}(s, v) \wedge \text{INTERCEPTEDSTORE}(s)$$

Second, we say a load l covered by a store s in a handler I cannot read from any store s' that cannot preempt I, because the value of s' will always be overwritten by s. That is, since s' cannot preempt I, it cannot execute in between s and l.

$$\text{MUSTNOTREADFROM}(l, s) \leftarrow \text{COVEREDLOAD}(l) \wedge \text{LOAD}(l, v) \wedge \text{STORE}(s, v) \wedge \text{NOPREEMPT}(s, l)$$

Third, we say that, if a store s is intercepted in a handler I, then a load l of the same variable that cannot preempt s, cannot read from the value stored by s. This is because the store intercepting s will always overwrite the value.

$$\text{MUSTNOTREADFROM}(l, s) \leftarrow \text{INTERCEPTEDSTORE}(s) \wedge \text{STORE}(s, v) \wedge \text{LOAD}(l, v) \wedge \text{NOPREEMPT}(l, s)$$

B. The Running Examples

To help understand how MUSTNOTREADFROM is deduced from the Datalog rules and facts, we provide a few examples. For ease of comprehension, we show in Table II how MUSTNOTREADFROM may be deduced from INTERCEPTEDSTORE, COVEREDLOAD and NOPREEMPT. Since all stores are either in or outside INTERCEPTEDSTORE, and all loads are either in or outside COVEREDLOAD, our rules capture the MUSTNOTREADFROM relation between all stores and loads.

TABLE II
MUSTNOTREADFROM RULES BASED ON INTERCEPTEDSTORE, COVEREDLOAD AND PRIORITY

	INTERCEPTEDSTORE(s)	NOT INTERCEPTEDSTORE(s)
COVEREDLOAD(l)	MUSTNOTREADFROM(l, s)	1) Not NOPREEMPT(s, l) → Possibly READFROM(l, s) 2) NOPREEMPT(s, l) → MUSTNOTREADFROM(l, s)
NOT COVEREDLOAD(l)	1) NOPREEMPT(l, s) → MUSTNOTREADFROM(l, s) 2) Not NOPREEMPT(l, s) → Possibly READFROM(l, s)	Possibly READFROM(l, s)

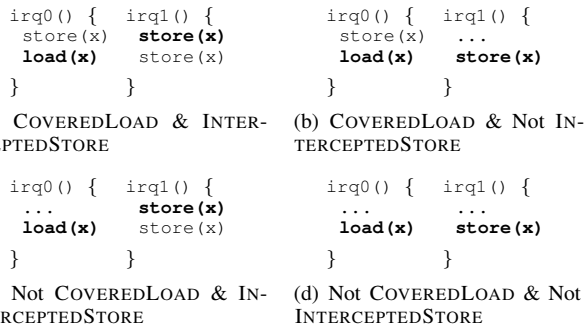
Specifically, if a store s is INTERCEPTEDSTORE and a load l is COVEREDLOAD, there is no way for the load to read from the store (Row 2 and Column 2).

If a load l is not COVEREDLOAD and a store s is not INTERCEPTEDSTORE, the load may read from the store by running sequentially or via preemption (Row 3 and Column 3).

If a load l is COVEREDLOAD, a store s is not INTERCEPTEDSTORE and the handler of the store can preempt the handler of the load, the load may read from the store through preemption (the first case at Row 2 and Column 3). However, if the handler of the store cannot preempt the handler of the load, it is impossible for the load to read from the store; in this case, the load always reads from the store in the same interrupt handler (the second case at Row 2 and Column 3).

Lastly, if a load l is not COVEREDLOAD, a store s is INTERCEPTEDSTORE, and the handler of the load cannot preempt the handler of the store, the load cannot read from

the store since the value of the store is always overwritten by another store in the same handler (the first case at Row 3 and Column 2). However, if the handler of the load can preempt the handler of the store, then the load can read from the store through preemption in between two stores (the second case at Row 3 and Column 2).



(a) CoveredLoad & InterceptedStore

(b) CoveredLoad & Not InterceptedStore

(c) Not CoveredLoad & InterceptedStore

(d) Not CoveredLoad & Not InterceptedStore

Fig. 7. Examples for each case in Table II.

Fig. 7 shows concrete examples of the four cases presented in Table II, where the figures correspond to the cases.

Fig. 7(a) represents the case at Row 2 and Column 2 in Table II and Fig. 7(b) represents the case at Row 2 and Column 3. In both programs, only the interference between bold-style statements are considered.

Fig. 7(a) shows an interference between CoveredLoad and InterceptedStore. Since the load in `irq0` is always overwritten by another store in it and a value of the first store in `irq1` is always updated by a store in it, the load cannot read a value from the store by preemption or running sequentially.

Fig. 7(b) shows an interference between CoveredLoad and not InterceptedStore. In this case, if `irq1` can preempt `irq0`, then the store from `irq1` can occur between the store and the load in `irq0`. Otherwise, the load from `irq0` cannot read a value from the store from `irq1`.

Fig. 7(c) shows an interference between not CoveredLoad and InterceptedStore. Similarly, if `irq0` can preempt `irq1`, the load from `irq0` can occur between the two stores from `irq1`. Thus, it is possible for the load to read a value from the first store in `irq1`. Otherwise, the load cannot read a value from the store by preemption of `irq1` or running sequentially.

Fig. 7(d) shows an interference between not CoveredLoad and not InterceptedStore. Here, the load in `irq0` can read a value from the store in `irq1` by running sequentially or through preemption. Therefore, it is possible for the load to read a value from the store as described at Row 3 and Column 3 in Table II.

To sum up, we can use the three inference rules to determine infeasible store-to-load pairs for all these cases.

C. Soundness of the Analysis

By soundness, we mean the MustNotReadFrom relation deduced from the Datalog facts and rules is an *under-approximation*. That is, any pair (l, s) of load and store in this relation is guaranteed to be infeasible. However, we do not attempt to identify *all* the infeasible pairs because the goal here is to *quickly* identify *some* infeasible pairs and skip them during the more expensive abstract interpretation computation.

Theorem 2: Whenever MustNotReadFrom(l, s) holds, the load l cannot read from the store s on any concrete execution of the program.

The soundness of our analysis as stated above can be established in two steps. First, assume that each individual rule is correct, the composition is also correct. Second, while presenting these rules, we have sketched the intuition behind the correctness of each rule. A more rigorous proof can be formulated via proof-by-contradiction in a straightforward fashion, which we omit for brevity.

Algorithm 3 Analysis of the entire program (cf. Alg. 2).

10: **function** Interf($T = \langle N, n_0, \delta\rangle$, $T' = \langle N', n_0', \delta'\rangle$, S)
11: $I \leftarrow \varnothing$
12: **for all** $n \in N$ **do**
13: **if** n is a shared memory write to variable v **then**
14: $I(v) \leftarrow I(v) \uplus \{(n, \text{TFUNC}(n, S(n))\}$
15: **return** I

Algorithm 4 Analysis of a single interrupt (cf. Alg. 1).

1: **function** AnalyzeLocal($T = \langle N, n_0, \delta\rangle, I$)
...
6: **if** n is a shared-memory read of variable v **then**
7: $i \leftarrow \bigsqcup\{n \mid (st, s) \in I(v) \wedge \neg\text{MustNotReadFrom}(l, st)\}$
8: $s \leftarrow \text{TFUNC}(n, S(n) \sqcup i)$
9: **else**
10: $s \leftarrow \text{TFUNC}(n, S(n))$
...

V. The Overall Analysis Procedure

We now explain how to integrate the feasibility checking technique into the overall procedure for iterative analysis, which leverages the MustNotReadFrom relation to improve performance. Specifically, when analyzing each interrupt handler T, we filter out any interfering stores from other interrupt handlers that are deemed infeasible, thereby preventing their visibility to T. This can be implemented in Algorithm 2 by modifying the function Interf, as well as the function AnalyzeLocal defined in Algorithm 1.

Our modifications to Interf are shown in Algorithm 3. That is, when computing the interferences of T, we choose to create a set of store–state pairs, instead of eagerly joining all these states. By delaying the join of these states, we obtain the opportunity to filter out the infeasible store-to-load pair individually. For this reason, we overload the definition of $\uplus$ to be the join ($\sqcup$) of sets on matching variables.

Next, we modify the abstract interpretation procedure for a single interrupt handler as shown in Algorithm 4. The process remains the same as AnalyzeLocal of Algorithm 1 except that, when a load l is encountered (Line 6), we join the state from all interfering stores while removing any that must not interfere with l, as determined by the MustNotReadFrom relation (Line 7).

The remainder of the modular analysis remains the same as in Algorithm 2.

```
irq0() {
  b = x;
  assert(b == 0);
}

irq1() {
  while(...) {
    x = 1;
    x = 0;
  }
}
```

Fig. 8. A small example with a loop.

For example, in Fig. 7(a), existing thread-modular abstract interpretation methods would consider the two stores from `irq1` for the load of `x` in `irq0`. In contrast, we use Algorithm 4 to remove the pairing of the load in `irq0` and the first store in `irq1`, since the load and the store satisfy the MUSTNOTREADFROM relation. Similarly, the pairing of the load of `x` in `irq0` and the store of `x` in `irq1` is filtered out when `irq1`'s priority is not higher than `irq0`'s priority as shown in Fig. 7(b).

Our method can handle programs with loops. Fig. 8 shows an example, which has two interrupt handlers where `irq1` has higher priority than `irq0`. Note that `irq0` loads `x` and stores the value into `b`. Since `x` is initialized to `0`, the handler checks whether the value of `b` is `0`. `irq1` has a loop containing two stores of `x`. First, it stores the value `1` and then the value `0`. Using traditional thread-modular abstract interpretation, we would assume that `x=1` and `x=0` are all possible stores to the load of `x` in `irq0`. This would lead to a bogus violation of the assertion in `irq0`.

However, in our analysis, this bogus violation is avoided by using the *post-dominate* relation between statements. Inside the while-loop of `irq1`, `x=0` post-dominates `x=1`, meaning that `x=0` always occurs after `x=1`. Therefore, using our Datalog inference rules presented in the previous section, we conclude that the store `x=1` cannot reach the load of `x` in `irq0`. Thus, it is impossible for the value `1` to be stored in `b` and then cause the assertion violation.

VI. EXPERIMENTS

We have implemented IntAbs, our new abstract interpretation framework in a static verification tool for interrupt-driven C programs. It builds on a number of open-source tools including Clang/LLVM [1] for implementing the C front-end, Apron library [16] for implementing the abstract domains, and μZ [13] for solving the Datalog constraints. We experimentally compared IntAbs with both iCBMC [18], a model checker for interrupt-driven programs and the state-of-the-art thread-modular abstract interpretation method by Miné [24], [27]. We conducted our experiments on a computer with an Intel Core i5-3337U CPU, 8 GB of RAM, and the Ubuntu 14.04 Linux operating system.

Our experiments were designed to answer the following research questions:

- Can IntAbs prove more properties (e.g., assertions) than state-of-the-art techniques such as iCBMC [18] and Miné [24], [27]?
- Can IntAbs achieve the aforementioned higher accuracy while maintaining a low computational overhead?
- Can IntAbs identify and prune away a large number of infeasible store-load pairs?

TABLE III
BENCHMARK PROGRAMS USED IN OUR EXPERIMENTAL EVALUATION.

Name	Description
test	Small programs created to conduct the sanity check of IntAbs's handling of various interrupt semantics.
logger	Programs that model parts of the firmware of a temperature logging device from a major industrial enterprise. There are two major interrupt handlers: one for measurement and the other for communication.
blink	Programs that control LED lights connected to the MSP430 hardware, to check the timer values and change LED blinking based on the timer values.
brake	Programs generated from the Matlab/Simulink model of a brake-by-wire system from Volvo Technology AB, consisting of a main interrupt handler and four other handlers for computing the braking torque based on the speed of each wheel.
usbmouse	USB mouse driver from the Linux kernel, consisting of the device open, probe, and disconnect tasks with interrupt handlers.
usbkbd	USB keyboard driver from the Linux kernel, consisting of the device open, probe, and disconnect tasks with interrupt handlers.
rgbled	USB RGB LED driver from the Linux kernel. We use initialization of *led* and *rgb* functions and the *led* probe function, and check the consistency of the *led* and *rgb* device values using interrupts.
rcmain	Linux device driver for a remote controller core, including operations such as device register, free, check the device information, and update protocol values. We check the consistency of the device information and protocol values using several interrupt handlers.
others	Programs collected from Linux kernel drivers for supporting hardware such as ISA boards, TCO timer for i8xx chipsets, and watch dog.

Toward this end, we evaluated IntAbs on 35 interrupt-driven C programs, many of which are from real applications such as control software, firmware, and device drivers. These benchmark programs, together with our software tool, have been made available online [14]. The detailed description of each benchmark group is shown in Table III. In total, there are 22,541 lines of C code.

A. Results

Table IV shows the experimental results. Columns 1-4 show the name, the number of lines of code (LoC), the number of interrupt handlers, and the number of assertions used for each benchmark program. Columns 5-7 show the results of `iCBMC` [18], including the number of violations detected, the number of proofs obtained, and the total execution time. Columns 8-10 show the results of Miné's abstract interpretation method [24], [27]. Columns 11-13 show the results of IntAbs, our new abstract interpretation tool for interrupts.

Since `iCBMC` conducts *bounded* analysis, when it detects a violation, it is guaranteed to be a real violation; however, when it does not detect any violation, the property remains undetermined. Furthermore, since `iCBMC` by default stops as soon as it detects a violation, we evaluated it by repeatedly removing the violated property from the program until it could no longer detect any new violation. Also note that since `iCBMC` requires the user to manually set up the *interrupt-enabled points* as described in [18], during the experiments, we first ordered the interrupts by priority and then set interrupt-enabled points at the beginning of the next interrupt handler. For example, given three interrupts `irq_L`, `irq_M` and `irq_H`, we would set the enabled point of `irq_L` in a main function, the enabled point of `irq_M` at the beginning of `irq_L`, and the enabled point of `irq_H` at the beginning of `irq_M`.

Overall, `iCBMC` found 88 violations while obtaining 0 proofs. `Miné`'s method, which was geared toward proving properties in threads, obtained 8 proofs while reporting 254 warnings,

TABLE IV
RESULTS OF COMPARING IntAbs WITH STATE-OF-THE-ART TECHNIQUES ON 35 INTERRUPT-DRIVEN PROGRAMS.

				iCBMC [18]			Miné [24], [27]			intAbs (new)		
Name	LOC	Interrupts	Assertions	Violations	Proofs	Time (s)	Warnings	Proofs	Time (s)	Warnings	Proofs	Time (s)
test1	46	2	2	0	0	0.23	1	1	0.18	0	2	0.07
test2	65	3	3	1	0	0.55	3	0	0.05	1	2	0.06
test3	86	4	4	1	0	0.52	4	0	0.06	2	2	0.10
test4	56	2	2	1	0	0.52	2	0	0.04	1	1	0.05
test5	54	2	2	1	0	1.56	2	0	0.04	1	1	0.04
logger1	161	2	1	0	0	0.45	1	0	0.22	0	1	0.27
logger2	183	3	3	0	0	0.50	2	1	0.29	0	3	0.39
logger3	195	4	4	0	0	0.46	1	3	0.31	0	4	0.43
blink1	164	3	3	1	0	0.65	3	0	0.12	2	1	0.18
blink2	174	4	3	1	0	0.67	3	0	0.16	2	1	0.30
blink3	194	5	4	2	0	1.14	4	0	0.25	3	1	0.46
brake1	819	2	5	1	0	0.87	3	2	0.66	1	4	0.98
brake2	818	3	4	3	0	2.24	4	0	1.67	3	1	1.91
brake3	833	4	5	2	0	2.38	5	0	2.58	4	1	3.48
usbmouse1	426	2	8	2	0	0.79	7	1	0.11	2	6	0.13
usbmouse2	442	4	16	2	0	0.69	16	0	0.31	5	11	0.69
usbmouse3	449	5	20	11	0	4.00	20	0	0.52	11	9	1.28
usbkbd1	504	2	8	3	0	0.91	8	0	0.23	4	4	0.39
usbkbd2	512	3	12	2	0	1.20	12	0	0.51	4	8	1.09
usbkbd3	531	5	20	3	0	1.19	20	0	1.86	12	8	4.44
rgbled1	656	2	10	5	0	0.71	10	0	0.41	5	5	0.77
rgbled2	679	3	15	5	0	1.11	15	0	0.99	5	10	2.39
rgbled3	701	4	20	5	0	1.07	20	0	2.18	10	10	5.68
rcmain1	2060	3	9	0	0	5.36	9	0	1.58	0	9	1.80
rcmain2	2088	5	15	6	0	12.39	15	0	6.93	6	9	9.46
rcmain3	2102	6	18	9	0	3.95	18	0	12.20	9	9	16.35
i2c_pca_isa_1	321	4	6	0	0	0.41	6	0	0.14	0	6	0.29
i2c_pca_isa_2	341	6	10	8	0	2.24	10	0	0.36	8	2	1.05
i2c_pca_isa_3	363	8	14	12	0	4.98	14	0	0.85	12	2	2.48
i8xx_tco_1	757	3	2	0	0	0.30	2	0	0.28	0	2	0.35
i8xx_tco_2	949	4	2	1	0	0.96	2	0	0.43	1	1	0.54
i8xx_tco_3	944	6	3	0	0	0.52	3	0	0.81	0	3	1.04
wdt_pci_1	1239	4	2	0	0	0.41	2	0	0.40	0	2	0.61
wdt_pci_2	1290	6	3	0	0	0.43	3	0	0.78	1	2	1.45
wdt_pci_3	1339	8	4	0	0	0.39	4	0	1.41	3	1	3.21
Total	22,541	136	262	88	**0**	56.75	(254)*	**8**	39.92	(118)	**144**	64.21

* indicates the results contain bogus warnings, because the technique was designed for threads, not for interrupts.

many of which turned out to be *bogus* warnings. In contrast, our new method, IntAbs, obtained 144 proofs while reporting 118 warnings. This is significantly more accurate than the prior techniques.

In terms of the execution time, IntAbs took 64 seconds, which is slightly long than the 39 seconds taken by `Miné`'s method and the 56 seconds taken by `iCBMC`.

B. Infeasible Pairs

Since IntAbs removes infeasible store-load pairs during the iterative analysis of individual interrupt handlers, it tends to spend extra time checking the feasibility of these data flow pairs. Nevertheless, this is the main source of accuracy improvement of IntAbs. Thus, to understand the trade-off, we have investigated, for each benchmark program, the total number of store-load pairs and the number of infeasible store-load pairs identified by our technique. Table V summarizes the results, where Column 3 shows the total number of store-load pairs, Column 4 shows the number of infeasible pairs, and Column 5 shows the percentage.

Overall, our Datalog-based method for computing the MUSTNOTREADFROM relation helped remove 69% of the load-store pairs, which means the subsequent abstract interpretation procedure only has to consider the remaining 31% of the load-store pairs. This allows IntAbs to reach a fixed point not only quicker but also with significantly more accurate results.

VII. RELATED WORK

We have reviewed some of the most closely-related work. In addition, Miné [28] proposed an abstract interpretation based technique for proving the absence of data-races, deadlocks, and other runtime errors in real-time software with dynamic priorities, which is an extension of his prior work [24], [26], [27] by adding priorities while targeting the OSEK/AUTOSAR operating systems. Specifically, it tracks the effectiveness of mutex, yield and scheduler state based on execution traces to figure out reachability, while using priorities to make the analysis more accurate. However, the technique may not be efficient in terms of memory and speed since it needs to check all mutex, yield, and scheduler state to determine spurious interference through trace history. Furthermore, it has not been thoroughly evaluated on practical benchmarks.

Schwarz and Müller-Olm [34] proposed a static analysis technique for programs synchronized via the priority ceiling protocol. The goal is to detect synchronization flaws due to concurrency induced by interrupts, especially for data

TABLE V
RESULTS OF TOTAL AND FILTERED STORE-LOAD PAIRS USING INTABS.

Name	LOC	# of Pairs	# of Filtered Pairs	Filtered Ratio
test1	46	1	1	100%
test2	65	4	2	50%
test3	86	16	8	50%
test4	56	4	3	75%
test5	54	4	3	75%
logger1	161	18	2	11%
logger2	183	32	6	18%
logger3	195	34	6	17%
blink1	164	19	15	78%
blink2	174	56	32	57%
blink3	194	120	63	52%
brake1	819	34	24	70%
brake2	818	82	58	70%
brake3	833	164	128	78%
usbmouse1	426	12	8	66%
usbmouse2	442	168	136	80%
usbmouse3	449	288	208	72%
usbkbd1	504	40	20	50%
usbkbd2	512	120	80	66%
usbkbd3	531	400	280	70%
rgbled1	656	76	38	50%
rgbled2	679	228	152	66%
rgbled3	701	456	304	66%
rcmain1	2060	84	84	100%
rcmain2	2088	560	476	85%
rcmain3	2102	840	714	85%
i2c_pca_isa_1	321	33	33	100%
i2c_pca_isa_2	341	210	110	52%
i2c_pca_isa_3	363	434	240	55%
i8xx_tco_1	757	14	12	85%
i8xx_tco_2	949	28	20	74%
i8xx_tco_3	944	39	33	84%
wdt_pci_1	1239	60	40	66%
wdt_pci_2	1290	150	82	54%
wdt_pci_3	1339	288	139	48%
Total	22,541	5,116	3,560	69%

races and transactional behavior of procedures. However, it is not a general-purpose verification procedure and cannot prove the validity of assertions. Regehr et al. [32] proposed to use context-sensitive abstract interpretation of machine code to guarantee stack-safety for interrupt-driven programs. Kotker and Seshia [17] extended a timing analysis procedure from sequential programs to interrupt-driven programs with a bounded number of context switches. As such, it does not analyze all behaviors of the interrupts. Furthermore, the user needs to come up with a proper bound of the context switches and specify the arrival time for interrupts.

Wu et al. [43] leveraged (bound) model checking tools to detect data-races in interrupt-driven programs. Kroening et al. [18] also improved the CBMC bounded model checker to support the verification of interrupt-driven programs. However, they only search for a bounded number of execution steps, and thus cannot prove the validity of assertions. Wu et al. [42] also proposed a source-to-source transformation technique similar to Regehr [31]: it sequentializes interrupt-driven programs before feeding them to a verification tool. However, due to the bounded nature of the sequentialization process, the method is only suitable for detecting violations but not for proving the absence of such violations.

This limitation is shared by testing methods. For example, Regehr [30] proposed a testing framework that schedules the invocation of interrupts randomly. Higashi and Inoue [12] leveraged a CPU emulator to systematically trigger interrupts to detect data races in interrupt-driven programs. However, it may be practically infeasible to cover all interleavings of interrupts using this type of techniques.

Wang et al. [39], [40] proposed a hybrid approach that combines static program analysis with dynamic simulation to detect data races in interrupt-driven programs. Although the approach is useful for detecting bugs, it cannot be used to obtain proofs, i.e., proofs that assertions always hold.

There are also formal verification techniques for embedded software based on model checking [11], [15], [33], [36]–[38], [44]. For example, Schlich and Brutschy [33] proposed the reduction of interrupt handler points based on partial order reduction when model checking embedded software. Vórtler et al. [36] proposed, within the Contiki system, a method for modeling interrupts at the level of hardware-independent C source code and a new modeling approach for periodically occurring interrupts. Then, they verify programs with interrupts using CBMC, which is again a *bounded* model checker. This means the technique is also geared toward detecting bugs and thus cannot prove properties. Furthermore, since it models the periodical interrupt invocation only, the approach cannot deal with non-periodic invocations.

Datalog-based analysis techniques have been widely used in testing and verification [2], [7], [9], [10], [21], [23], [29], [41], but none of the prior techniques was designed for analyzing the interleaving behavior of interrupts. For example, Kusano and Wang [19], [20] used Datalog to obtain flow-sensitivity in threads to improve the accuracy of thread-modular abstract analysis for concurrent programs. Sung et al. [35] used Datalog-based static analysis of HTML DOM events to speed up a testing tool for JavaScript based web applications. However, the Datalog rules used by these prior techniques cannot be used to reason about the behavior of interrupts with priorities.

VIII. CONCLUSIONS

We have presented an *abstract interpretation* framework for static verification of interrupt-driven software. It first analyzes each individual handler function in isolation and then propagates the results to other handler functions. To filter out the infeasible data flows, we have also developed a constraint-based analysis of the scheduling semantics of interrupts with priorities. It relies on constructing and solving a system of Datalog constraints to decide whether a set of data flow pairs may co-exist. We have implemented our method in a software tool and evaluated it on a large set of interrupt-driven programs. Our experiments show the new method not only is efficient but also significantly improves the accuracy of the results compared to existing techniques. More specifically, it outperformed both iCBMC, a bounded model checker, and the state-of-the-art abstract interpretation techniques.

IX. ACKNOWLEDGMENTS

This material is based upon research supported in part by the U.S. National Science Foundation under grants CCF-1149454 and CCF-1722710 and the U.S. Office of Naval Research under award number N00014-17-1-2896.

REFERENCES

[1] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *ACM/IEEE international symposium on Microarchitecture*, 2003.

[2] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 243–262, 2009.

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

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 168–176, 2004.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *European Symposium on Programming Languages and Systems*, pages 21–30, 2005.

[7] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–308, 2012.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference On Automated Software Engineering*, 2016.

[10] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[11] Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 326–336, 2017.

[12] Makoto Higashi, Tetsuo Yamamoto, Yasuhiro Hayase, Takashi Ishio, and Katsuro Inoue. An effective method to control interrupt handler for data race detection. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 79–86, 2010.

[13] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462, 2011.

[14] The intAbs tool and benchmark programs for evaluating intAbs. URL: https://github.com/sch8906/intAbs.

[15] Franjo Ivančić, I. Shlyakhter, Aarti Gupta, M.K. Ganai, V. Kahlon, Chao Wang, and Z. Yang. Model checking C program using F-Soft. In *International Conference on Computer Design*, pages 297–308, 2005.

[16] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667, 2009.

[17] Jonathan Kotker, Dorsa Sadigh, and Sanjit A. Seshia. Timing analysis of interrupt-driven programs under context bounds. In *International Conference on Formal Methods in Computer-Aided Design*, pages 81–90, 2011.

[18] Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with nested interrupts. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 229–234, 2015.

[19] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 799–809, 2016.

[20] Markus J. Kusano and Chao Wang. Thread-modular static analysis for relaxed memory models. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 337–348, 2017.

[21] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–12, 2005.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[23] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[24] Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Programming Languages and Systems*, pages 398–418. 2011.

[25] Antoine Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *Proc. of the 10th School of Modelling and Verifying Parallel Processes*, pages 35–48, 2012.

[26] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), 2012.

[27] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 39–58, 2014.

[28] Antoine Miné. Static analysis of embedded real-time concurrent software with dynamic priorities. *Electr. Notes Theor. Comput. Sci.*, 331:3–39, 2017.

[29] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[30] John Regehr. Random testing of interrupt-driven software. In *International Conference on Embedded Software*, pages 290–298, 2005.

[31] John Regehr and Nathan Cooprider. Interrupt verification via thread verification. *Electron. Notes Theor. Comput. Sci.*, 174(9):139–150, 2007.

[32] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.*, 4(4):751–778, 2005.

[33] Bastian Schlich, Thomas Noll, Jörg Brauer, and Lucas Brutschy. Reduction of interrupt handler executions for model checking embedded software. In *Haifa Verification Conference*, pages 5–20, 2009.

[34] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich, and Markus Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 93–104, 2011.

[35] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static DOM event dependency analysis for testing web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 447–459, 2016.

[36] Thilo Vörtler, Benny H'"ockner, Petra Hofstedt, and Thomas Klotz. Formal verification of software for the Contiki operating system considering interrupts. In *IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 295–298, 2015.

[37] Chao Wang and Kevin Hoang. Precisely deciding control state reachability in concurrent traces with limited observability. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 376–394, 2014.

[38] Chao Wang, Z. Yang, Franjo Ivancic, and Aarti Gupta. Disjunctive image computation for emebedded software verification. In *Proceedings of the Design, Automation & Test in Europe Conference*, 2006.

[39] Yu Wang, Junjing Shi, Linzhang Wang, Jianhua Zhao, and Xuandong Li. Detecting data races in interrupt-driven programs based on static analysis and dynamic simulation. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, pages 199–202, 2015.

[40] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. Automatic detection and validation of race conditions in interrupt-driven embedded software. In *International Symposium on Software Testing and Analysis*, pages 113–124, 2017.

[41] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[42] Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong, and Ji Wang. Static analysis of runtime errors in interrupt-driven programs via sequentialization. *ACM Trans. Embedded Comput. Syst.*, 15(4):70:1–70:26, 2016.

[43] Xueguang Wu, Yanjun Wen, Liqian Chen, Wei Dong, and Ji Wang. Data race detection for interrupt-driven programs via bounded model checking. In *International Conference on Software Security and Reliability*, pages 204–210, 2013.

[44] Z. Yang, Chao Wang, Franjo Ivančić, and Aarti Gupta. Mixed symbolic representations for model checking software programs. In *International Conference on Formal Methods and Models for Codesign*, pages 17–24, 2006.

[45] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.