Rchecker: A CBMC-based Data Race Detector for Interrupt-driven Programs

Haining Feng[1], Liangze Yin[1], Wenfeng Lin[2], Xudong Zhao[1], Wei Dong[1]
[1]*Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, Changsha, China*
[2]*China Academy of Engineering Physics, Mianyang, China*
hainingbaby@gmail.com, yinliangze@nudt.edu.cn, linwenfeng_job@163.com, {zhaoxudong13, wdong}@nudt.edu.cn

***Abstract*—Interrupt-driven programs are widely used in aerospace, medical equipment, and other embedded systems that require extreme safety and stability. However, uncertain interrupt interleaving executions may cause serious data race problems. Static analysis is an important technology to detect data race problems. Existing methods are either too conservative to have low accuracy, or bring lots of false alarms, there still need a more effective solution. The program verification tool CBMC has an excellent performance in the analysis and verification of C multi-threaded modeling, but it doesn't support interrupt-driven programs. In this paper, we proposed a method based on CBMC to detect data race in interrupt-driven programs. Our method achieved accurate analysis of interrupt-driven programs by performing analysis toward interrupt preemption and synchronization semantics, and further validation of the program can be supported. Experiments results on related benchmarks demonstrate our approach's usability and effectiveness.**

***Index Terms*—interrupt-driven programs, data race, static analysis, bug detection**

I. Introduction

Interrupt-driven program is essential in aerospace, weapon systems, and automotive electronics. The main task continuously responds to the interrupt services and leverages interrupt function to finishing system works. Plenty of engineering practice and experience shows that this kind of programming paradigm can lead to serious data race problems. Data race can make the operating system kernel crash and the missile blast point shift, most of the problems discovered in the spacecraft software during the test phase are related to data race, the harm data race caused is huge.

In the interrupt-driven program, shared variables refer to those variables that read/write access to them simultaneously exists in the main task and at least one interrupt. and messaging between concurrent flows is mostly realized by a large number of shared variables, which may cause data race easier and detection results may have many false alarms. At the same time, uncertain interleaving execution creates obstacles to finding data race precisely.

Static analysis is an important way to find interrupted data race, but because it usually ignores happens-before principles in the program and lack of runtime information, most of the static data race detection methods may either have plenty of false alarms or miss many real errors.

Corresponding author: Liangze Yin. This work was funded by the National Nature Science Foundation of China (No. 61802415, No. 61690203, No. 61532007).

CBMC [1] is a program verification tool for multi-threaded programs, it will follow program paths to an assertion, build a formula that corresponds to the path, solved by an SAT/SMT solver to finally obtain a satisfying *static single assignment* (SSA) to transfer program as formulas. CBMC performs well on multi-threaded programs in practice, however, it doesn't support interrupts yet.

This paper focuses on interrupt-driven programs based on shared variables and inter-procedure analysis. Based on CBMC, we achieved static analysis towards interrupt-driven program data race problems and accurately describe and analysis interrupted semantics of preemption and synchronization. We perform precise analysis towards interrupt-driven programs, furthermore, we can use the back-end of CBMC to perform subsequent analysis and verification if necessary.

We have implemented our method on the top of CBMC and applied it to the related benchmarks. The result shows that our method finds all data race problems in benchmarks within a short time. In summary, this paper makes the following contributions:

- We propose a sound and precise static analysis method based on CBMC, which will automatically process interrupt-driven programs and detect data race problems efficiently.
- Our method support analysis to interrupted preemption and synchronization semantics. We use priority theory to express preemption and proposed interrupt mask lists (IML) to process synchronization, which is a key to reduce false alarms of detection technology.
- We implement our method in CBMC and developed a prototype named Rchecker to detect data race problems in the interrupt-driven programs. We evaluated it on the benchmarks which are designed intentionally with data race problems. The result indicates that our method finds all races in a short time with fewer false alarms.

The rest of this paper is organized as follows. Section II discusses the features and sematics of interrupt-driven program. Section III give a detailed description of our work. The experiment evaluation is given in Section IV. Finally, we discuss related work in Section V and conclusion in Section VI.

II. Preliminaries

A. Interrupt-driven Program

An interrupt-driven program consists of one main task and $N \geq 1$ interrupts, performed by the main task continuously responding to the interrupts. It contains a set of global shared variables as well as local variables.

Concurrency. In interrupt-driven programs, interrupts happen randomly which means it is unpredictable. Considering the similarity between multi-threaded and interrupts is that, new concurrent flows (threads or interrupts) may arrive at any time during program execution if there is no lock protection, to express concurrency, we simulate interrupted concurrency as multi-threaded does, and create consistent corresponding statements for interrupts at the beginning of the program code.

Preemption. Interrupts have priority, the interrupt owns high priority can preempt low, but lower interrupt can not preempt higher in turn. The preemption between task and interrupt, interrupt and interrupt is asymmetric. We assume that there is no nested phenomenon in the program. Nesting happens when there exist several interrupt preemptions simultaneously which makes system behaviors extremely complicated, it can be infinite theoretically. However, considering the feature that interrupts will return after its execution, so regardless of how deep the nesting depth is, analyzing on preemption comprehensively is enough.

Synchronization. Non-standard low-level synchronization mechanisms (such as disable-enable interrupts, suspend the scheduler, and flag-based synchronization) is a typical way to achieve concurrency control in interrupt-driven programs. We assume that options on interrupt mask register `diable_isr` and `enable_isr` are used to disable and enable interrupts, and proposed interrupt mask list to process synchronization of interrupt-driven program.

B. Vulnerability type

In this paper, we investigate existing research and refine data race defect patterns for interrupt-driven programs. We focus on two types of data race: multibyte variable conflict and access sequence conflict.

Multibyte variable conflict. When C programming language compiled into the underlying machine instructions, a single C statement often corresponds to multiple machine instructions. Therefore, reading or writing to a multi-byte variable is divided into several steps instead of atomic. As shown in Fig. 1.

A read operation to the variable `multibyte` occurs in a low-priority concurrent flow (task function) and there exits a write operation `++multibyte` in interrupt function. If interrupt service triggered when the assembly instructions execute halfway, then there is a potential data race. Traditionally, data race happens when two or more threads access the same memory location concurrently, and at least one of the access

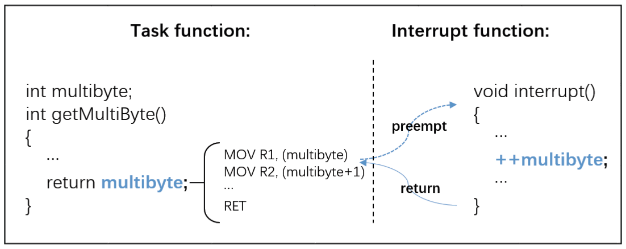

Fig. 1. Multibyte variable conflict pattern.

is a write. We attach one more condition, that is single access to the variable in assembling is not atomic.

Access sequence conflict. The race problems is related to three access locations rather two, for the reason that in interrupt, data race is more related to disrupte program's expected behavior. Consider example shown in Fig. 2.

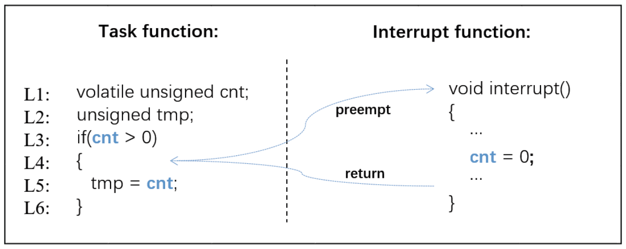

Fig. 2. Access sequence conflict.

In task, *line 5* and *line 3* are expected to get the same value of `cnt`, if interupt comes between them, then the write operation to `cnt` in interrupt will change its value and disrupte the expectation, then data race problem arise .

We let access sequence be a triple$< a_1, a_2, a_3 >$, where a_1, a_3 is two consecutive accesses in low-priority concurrent flow and a_2 comes from high-priority. There are 8 permutations among three read/write events. Some permutations are harmless such as $< R, R, R >$, simultaneous reading has no impact on the program, and we exclude it from consideration. Access sequence conflict has been summarized into four types as follows:

- $< R, W, R >$: Access a_1, a_3 is expected to read the same value, while a_2 brokes hypothesis.
- $< W, W, R >$: Access a_3 is expected to read assignment result of a_1, when a_2 comes from interupt, as a result, a_3 read variable value writen by a_2 rather expected a_1.
- $< R, W, W >$: Typical situation is that a_1 is related to branch condition, while a_3 is write operation under the branch. and a_2 is inappropriate at this time.
- $< W, W, W >$: In this case, the race happens when continuous writing to the same data elements a_1, a_3 (like array, buffer, etc.) encounters sudden write a_2.

C. CBMC

Bounded Model Checking [2] is the most successful formal validation technique to verify the system's behavior, especially when the system state space becomes complicated. CBMC is a useful BMC tool for multi-threaded C programs. We use CBMC as a front-end to parse interrupt-driven program. In general, CBMC has the following advantages:

- Handle complex data structures, like a multidimensional pointers, dynamic array allocation, etc.
- Perform precise pointer alias analysis.
- Supports almost all C language grammar and semantics.
- Parse and get all shared variables appeared in multi-threaded programs exhaustively.
- The back-end of CBMC can support further analysis and verification based on existing work.

We assume that loops in the program have limited unwinding depths which is a basic principle in CBMC, and data race defects can be detected under limited loops. We use modeling about C language data elements like pointers, structures, and unions, etc., as the same as CBMC does, for they are independent of interrupt-driven program analysis and CBMC has done a good job in this regard. We perform analysis based on its memory model and SSA.

III. Algorithms and Illustration

The overview of our work is shown in Fig. 3. Given an interrupt-driven C program, we first obtain interrupt-related information, such as interrupt function name and its priority, those messages should be identified by specific external configuration. We distinguish interrupt by external configuration information, then parse the program and shared access space is built for the program, which is defined as follows:

***Definition** 1 (Shared access space):* Shared access space is a finite set of all shared access events that appear in the program. Shared access means a single read/write option to the certain memory address (like variables, ports, registers) in the program, and the access is global rather local, otherwise, it won't be shareable. We define $S = (A, T, P, I_M)$ be a shared access space where:

- A is a set of program memory address accessed.
- The set of T is the set of concurrent flows, i.e., interrupts.
- Every interrupt has its own priority, all priorities make up the set P.
- I_M represents a list for interrupt mask status, it contains a list of integers and indicates which priorities of interrupt in the current program location are currently masked. Where **interrupt mask list** $IML \subset I_M$ is maintained by each single shared access in the critical section.

We use CBMC to extract all shared read/write access to construct shared access space. To make the best use of this space, we divide the entire space into many basic access blocks. Within basic access blocks, we build a local control flow graph to achieve less false positives and less false negatives. Among basic access blocks, we perform pattern matching to find data race problems in interrupts.

According to past detection or testing technology toward interrupt-driven programs, the most challenging problem is that it is not easy to accurately describe the specific semantics of preemption and synchronization mechanism, because system behavior is unpredictable and program path is too elusive to simulate it.

We employ interrupt-driven semantics to guide static analysis and incorporate semantic features into the analysis procedure. Specifically, preemption guides analysis between basic blocks, only high-priority basic access blocks can preempt low-priority, and the reverse is not true. Synchronization is characterized by dynamically maintaining interrupt mask lists (IML) in the location of programs where shared access happens. In this way, we take semantics into consideration and finally get precise results.

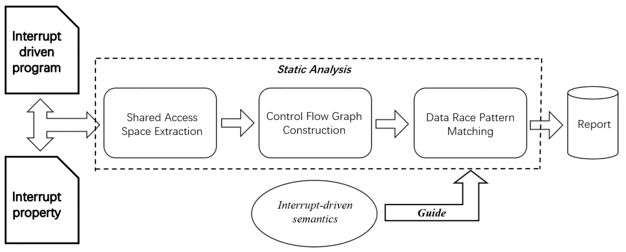
Interrupt driven program
Interrupt property
Static Analysis
Shared Access Space Extraction
Control Flow Graph Construction
Data Race Pattern Matching
Report
Interrupt-driven semantics
Guide

Fig. 3. An overview of our method.

A. Shared access space construction

We transfer program to SSA, from where extracts all shared read/write access to memory addresses and constructs shared access space, as shown in Fig. 4. Divided by memory address and concurrent flow, we derived the **basic access block**, i.e., all shared *read/write* access operations to single variable A in single concurrent flow T. In basic access blocks, we build control flow, and outside the blocks perform pattern matching of data race problems.

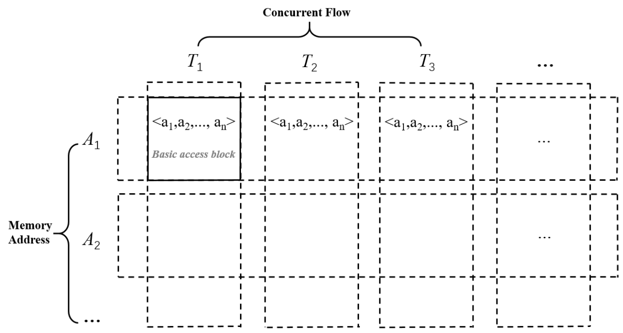

Fig. 4. Shared access space of interrupt-driven program.

Divided access blocks by T is necessary, for the reason that data race only happens between concurrent flows rather single flow, if all access to a memory address only occurs in single T, there is no data race at all. As a result, divided by memory address A can facilitate targeted analysis as well as improve analysis efficiency, because once all basic access blocks to single A only exist in single T, there is no necessary to investigate whether defect happens in this location.

B. Build control flow graph

To implement a flow-sensitive static analysis, we build the local control flow graph (LCFG) for basic access blocks. In basic access block, we get a collection of shared access events, it is in the order of appearance sequence in codes rather program path.

We use simple constraint solving to build LCFG, each shared access in the program has a path condition, we call it a ***guard***. For the code `If(A) Then B`, the guard of `B` is `A`. Suppose that the guard of access a_1, a_2 is g_1, g_2 respectively, we use off-the-shelf SMT solver to solve the united guard. If $g_1 \wedge g_2$ is satisfiable, then there may exist an edge between a_1 and a_2, if it is unsatifiable, then a_1 and a_2 belong to different branches and can not happen one after another. In this way, we can easily get all possible edges based on absolute happen-before relationship.

One serious problem is, there still have some possible edges from access events that unrelated in order of appearance, where and how deep we should backtrack to find all edges is the key point to build precise LCFG. To solve this, we propose the ***upper bound*** relationship of two shared access events.

***Definition** 2 (Upper bound, $\sup_R$):* For two shared accesses a_1, a_2 and its guard g_1, g_2, we say a_1 is upper bound of a_2 when a_1 is the earliest event that can happen continuously before a_2.

Suppose that `solve(`$g_1 \wedge g_2$`)` is the constraint solving result of $g_1 \wedge g_2$ where SAT means satifiable and UNSAT means unsatifiable, then we have theory as follows:

`solve(`$\neg g_1 \wedge g_2$`)` = UNSAT $\models a_1 = \sup_R(a_2)$

In a narrow sense, $g_1 \wedge g_2$ represents a program path from g_1 to g_2, $\neg g_1$ indicates an opposite path condition to g_1. When `solve(`$\neg g_1 \wedge g_2$`)` is SAT, it means that there exist a program path from $\neg g_1$ to g_2, and a_2 can happen without following the path where a_1 is. On the contrary, if it is UNSAT, it turns out that there exists no path that won't via a_1 to a_2, and there the upper bound of a_2 is a_1.

C. Data race pattern matching

To find data race problems by pattern matching, firstly we let CBMC process interrupts as threads because CBMC doesn't support interrupts at all. We create interrupted concurrent flow statements at the beginning of the program code just like multi-threaded programming did, and then take priority into consideration to process preemption. In order to highlight the preemption relationship, we let every concurrent flow T maintains a tuple of (T_{id}, F, B, P) where,

- Every T has a T_{id} as unique identification.
- F is function name of a task or interrupt.
- Every T has a set of basic access blocks B.
- P represent the priority of T.

From external interrupt configuration, we can extract all concurrent flows as tuples and rank the interrupt functions in descending order of priority. Consider that function name F and priority P among different T may be identical, so we use T_{id} as key and give them T_{id} in order.

When analyzing data race problem occurs between two concurrent flows T_1 and T_2, suppose that problem is caused by T_2 interrupting the execution of T_1 (like access sequence conflicts), if the priority of T_2 is lower than T_1's, the problem won't happen because preemption doesn't hold.

In this way, we enumerate all the possibilities among two concurrent flows, and according to the vulnerability types, we rule out failed preemption to complete the pattern matching process.

D. Interrupt semantic guidance

Weak support for synchronization may bring many false positives, synchronization prunes out many branches of programs that exactly not exist. We don't combine synchronization with the program's control flow information but proposed interrupt mask list (IML) for every shared access event in the critical section.

We use IML to describe synchronization of interrupts, identify duplicate `lock-unlock` problems simply and efficiently, and remove failed protection for shared accesses to expose potential data race defects. IML consists of an ordered collection of integers, every shared access in the area between the critical section maintains an IML. For example, $< 3, 2 >$ represent current access firstly disables interrupt with priority 3, then disables priority 2, if any other interrupts whose priority is neither 3 nor 2 and perform enable operation on priority 3 or 2, then there will break expected protection, we need to remove failed locks and update related IML. If the above does not happen, it means the protection is effective, and we just use IML to judge whether the race problems will happen.

Algorithm 1 Algorithm of updating IML

```
1: construct the initial IML for every shared access events
   when parsing programs;
2: $L \leftarrow$ IML of a variable from low-priority interrupt;
3: $H' \leftarrow$ set of IMLs of the same variable from higher-
   priority interrupt than $L$;
4: for each IML $H$ in $H'$ do
5:     acquire the priority $P_h$ that $H$ belonging;
6:     for each elements $e$ in $L$ do
7:         if $e == P_h$ then
8:             break;
9:         end if
10:        for any element $e'$ in $H$ do
11:            if $e == e'$ then
12:                remove $e$ from $L$;
13:            end if
14:        end for
15:    end for
16: end for
```

The initial IML for every shared access event in the critical section is constructed when parsing the program. When a disabled or enabled operation to interrupts detected, we push or pop the corresponding priority into current IML, after updating every IML, we finally get accurate IML to present which interrupts are disabled before current shared access happens. The IML describes synchronization of interrupts, at the same time solve the duplicate synchronization operation problems, it is an important factor in achieving accurate detection.

IV. Empirical Evaluation

We choose the Racebench 2.1 benchmarks which are available online [1] to evaluate our tool, many studies on interrupt-driven program detection have performed their experiments on this suite.

The Racebench consists of 30 cases and contains 50 bugs that insert manually, they are written according to realistic settings and reflects most of the syntactic and semantic features of C. The cases usually are small in size but not toy case, they are carefully constructed and have a detailed description about the program to help verify the correctness of related researches. Those cases have one or more data race problems, 1 to 4 interrupts, varied data structures and hundreds even thousands of read/write access events, which all make analysis on them become challenging.

Our experiments were conducted on PC with 2.2GHz Intel(R) i7-8750H CPU (four cores), 4GB RAM, and OS of Ubuntu16.04. For the loop unwinding depth of the program, the deeper the unwinding depth is, the more precise result we get, and the time consumption become higher in the meantime. We reduced the number of loops in the Racebench programs by a factor of 100 and set loop unwinding depth to fixed 101 times to balance accuracy and efficiency.

For all techniques, Table.I summarized the results of running our tool on Racebench benchmarks. Notice that one of the cases in Racebench encountered an unexpected error with our tool and we ignore it in the experiment. Among all the 30 cases, the size of the program range from 30 to 90 SLOC, and the average size of programs is 55 SLOC. The cases are shown in table.I are arranged in ascending of "Size" and then "Events". The column "Events" shows the number of shared *read/write* access events in each case, and the "Interrupts" column indicates the number of interrupts in the case. The detected result of Rchecker is shown in column "Detected Race" which consists of three parts, the "*total*" is a total number of data race problems that have been detected, *#CF* means the number of races in results that related to local control flow construction in benchmarks, and *#IS* is the number of races which is related to interrupt semantic guidance strategy. The run time for every case can be observed in the column "Time". Since the benchmarks have detailed descriptions including bug location, We compared our detected races and real races in the program, and use the column "Hit" to present how many percentages that we had successfully found races.

As we can see in Table.I, for every case, we found all data race problems that program actually exits with less false positives. The number of shared access events of each case ranges from 6 to 40.4k, and an increase in the scale of events will not lead to a significant increase in time consumption. The vast majority of cases are detected within a second, even for 40.4k shared access events, the analysis time to exhaust all state space is still quite acceptable.

From the experiment results above, we can conclude that our approach can effectively detect data race defects for interrupt-driven programs in an efficient way. We design two algorithm strategies for more effective detection in Section III, one is our local control flow construction strategy, and Algorithm.1 is the strategy of interrupt semantic guidance. Notice that interrupt semantics we have considered include interrupted concurrency, preemption and synchronization. We use synchronization and preemption as the basis of our analysis procedures, so all detected race results are under the influence of interrupt semantics. Here we focus on analyzing the performance of interrupt synchronization guidance strategy on the results.

To measure the performance of two strategies, we divided the results of detected races into two parts in Table.I. *#CF* and *#IS* is the number of races in detect results that related to our two strategies respectively. In benchmarks,some cases designed intentionally to create barriers to analyzing the control flow and synchronization relationship of the program. To determine whether we find real races or not, we pick it up and compared the *#CF* and *#IS* in detected races and real races one by one, then the benefits of the two strategies are determined. As shown in Table.II, with our strategies, we found all races related to LCFG and synchronization in programs, and these races each account for 40.8% and 32.7% of all data race problems in Racebench, which means that the two strategies improve tool's performance by 40.8% and 32.7% respectively.

From table.II we can see that both "#CF" and "#IS" can significantly improve the performance of our tool, and as the size of the program increases, the time consumption didn't increase too much. This indicates both the strategies of building the local control flow graph and interrupt semantics guiding have important contributions in terms of accuracy and efficiency.

V. Related Work

Savage [3] et al. proposed a dynamic data race detection tool *Eraser* based on the lockset algorithm, which maintains the current lock information of each thread during program execution, and updates the lock held by shared variables when the variable is no longer locked. Von [4] and Elmas [5] refined and extended the Eraser method, which enables more accurate and effective detection of object-level data race. Besides, Valgrind [6] integrates several tools together to find memory management and threading bugs. Helgrind [7] is part of it used to find race condition in multi-threaded programs and it is implemented on the top of *Eraser*.

Lamport [8] proposed happens-before relationship, based on it, Netzer [9] and Perkovic [10] use logical clocks to

[1]https://github.com/chenruibuaa/racebench/tree/master/2.1

TABLE I
EXPERIMENTAL RESULTS OF DETECTING INTERRUPT-DRIVEN CASES

Case ID	Size(LOC)	Events	Interrupts	Detected Race			Real Race			Time(s)	FP	Hit(%)
				total	#CF	#IS	total	#CF	#IS			
1	34	6	1	3	0	0	3	0	0	0.36	0	100
2	35	9	1	1	0	0	1	0	0	0.31	0	100
3	39	8	1	1	0	0	1	0	0	0.31	0	100
4	40	7	1	2	0	0	2	0	0	0.30	0	100
5	41	15	1	1	1	0	1	1	0	0.31	0	100
6	42	804	1	5	5	0	5	5	0	1.81	0	100
7	44	7	2	1	0	1	1	0	1	0.28	0	100
8	44	32	1	1	0	0	1	0	0	0.29	0	100
9	46	810	2	2	1	2	1	1	1	0.47	1	100
10	47	40.4k	1	3	1	0	1	1	0	5.20	2	100
11	48	29	1	2	0	0	2	0	0	0.34	0	100
12	49	9	3	2	0	2	2	0	2	0.32	0	100
13	51	14	1	11	4	0	1	1	0	0.37	10	100
14	53	218	1	2	0	0	1	0	0	0.32	1	100
15	54	12	3	2	0	2	1	0	1	0.31	1	100
16	54	16	1	1	0	0	1	0	0	0.32	0	100
17	54	271	1	1	1	0	1	1	0	0.41	0	100
18	55	18	2	3	3	3	2	2	2	0.32	1	100
19	57	12	3	2	0	2	1	0	1	0.35	1	100
20	60	517	3	2	1	2	1	1	1	0.36	1	100
21	65	22	2	3	0	0	3	0	0	0.30	0	100
22	65	520	1	2	0	0	1	0	0	0.36	1	100
23	67	515	3	2	1	2	1	1	1	0.45	1	100
24	67	516	1	6	2	0	4	2	0	0.34	2	100
25	69	37	2	3	0	3	1	0	1	0.31	2	100
26	72	31	1	2	2	2	1	1	1	0.37	1	100
27	74	2.2k	2	2	0	2	1	0	1	0.51	1	100
28	81	535	1	3	2	0	3	2	0	0.35	0	100
29	92	32	1	3	0	3	3	0	3	0.31	0	100
30	95	547	1	1	1	0	1	1	0	1.40	0	100
SUM				75	25	26	49	20	16		26	

TABLE II
STATISTICS OF PERFORMANCE OF TWO STRATEGIES

Strategy	Find Race[1]	Total Race	Improve
#CF	20	49	40.8%
#IS	16	49	32.7%

[1] The column "Find Race" refers to the overlapping part of "Detected Race " and "Real Race" in Table.I.

detect data race. The Djit+ [11] algorithm uses the vector-clock format to record the logical clock accessed by the thread and the shared memory, and only the first read/write access to the shared memory location is recorded in each time frame. FastTrack [12] replace heavyweight vector clocks with an adaptive lightweight representation and significantly improves time and space performance with no loss in precision. Loft [13] analysis some scenarios based on FastTrack to further reduce vector-clock based assignment and comparison operations. iFT [14] simplified FastTrack in some ways, it eliminates the need to traverse every item of vector-clock for analysis, but only focuses on left-most and right-most which reduces the detection complexity of read and write access operations to *O(1)*.

MultiRace [11] firstly utilized the improved lockset algorithm to find possible data race, and detect whether the feasible statement pairs are truly concurrent. Helgrind+ [15] constrains the sequence of operations executed by threads in a single segment, which reflects the logical clocks of threads. Thread-Sanitizer [16] use segment to represent consecutive memory access events in a thread, and made a good performance in practice. There are also some dynamic approaches using sampling technology such as LiteRace [17] and AtexRace [18].

There have been few visible results in the detection of data race for interrupt-driven programs. Regehr [19] proposed a way to verify the correctness of interrupt-driven program concurrency, i.e., first convert the program to thread-based program and then use the existing technology like YOGAR-CBMC [20] for multi-threaded program concurrent correctness verification. The approach is not rigorous, and further formal semantic representations are needed to ensure the correctness of the conversion. Mercer and Jones [21] propose a method for interrupt-driven embedded code model verification in conjunction with the state saving and recovery capabilities of a GDB debugger. SLAM [22] and RacerX [23] can also be used to check kernel code containing interrupts, but both are based on a specific interrupt call pattern and are not oriented towards interrupt-driven programs.

Wu [24] converts interrupt-driven programs to non-deterministic sequential programs then collect the path conditions under where race occurs. On this basis, the path conditions are converted to SMT formulas by a bounded model checking, by determining whether the path formula can be satisfied to eliminate false positives. Chopra [25] defines a natural notion of data races and a happens-before ordering for interrupt-driven programs, and proposed the notion of disjoint blocks to define the synchronization as well as efficient "sync-CFG" to carry static analysis. And Pan [26] proposes a new modeling language that extended the UML sequence diagram with interrupt fragment, mask variable, and task constraint to model the unpredictable and preemptive interrupt-driven behavior, guiding off-the-shelf tools and get good performance.

VI. Conclusion

In this paper, we propose an entire framework based on CBMC to detect data race problems in the interrupt-driven program. We implement analysis towards interrupts using CBMC and express interrupt preemption by priority of the concurrency. To obtain accuracy and efficiency, we build local CFG of programs to perform flow-sensitive analysis and proposed IML to describe the synchronization of the interrupt-driven program. The experiments on a set of programming tasks show that our approach can detect data race accurately and efficiently for the interrupt-driven program.

References

[1] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[4] C. von Praun and T. R. Gross, "Object race detection," *SIGPLAN Not.*, vol. 36, no. 11, pp. 70–82, Oct. 2001.

[5] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: Efficiently computing the happens-before relation using locksets," 01 2006, pp. 193–208.

[6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[7] "Valgrind project. Home of Memcheck, Helgrind and DRD," http://www.valgrind.org/.

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

[9] R. H. B. Netzer, "Pace condition detection for debugging shared-memory parallel programs," Ph.D. dissertation, Madison, WI, USA, 1991, uMI Order No. GAX91-34338.

[10] D. Perkovic and P. J. Keleher, "Online data-race detection via coherency guarantees," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. SI, pp. 47–57, Oct. 1996.

[11] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," *SIGPLAN Not.*, vol. 38, no. 10, pp. 179–190, Jun. 2003.

[12] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 121–133, Jun. 2009.

[13] Y. Cai and W. K. Chan, "Loft: Redundant synchronization event removal for data race detection," pp. 160–169, Nov 2011.

[14] O.-K. Ha and Y.-K. Jun, "An efficient algorithm for on-the-fly data race detection using an epoch-based technique," *Sci. Program.*, vol. 2015, pp. 13:13–13:13, Jan. 2015.

[15] V. W. A.Jannesari, Kaibin Bao, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–13.

[16] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71.

[17] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: Effective sampling for lightweight data-race detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 134–143, Jun. 2009.

[18] Y. Guo, Y. Cai, and Z. Yang, "Atexrace: Across thread and execution sampling for in-house race detection," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 315–325.

[19] J. Regehr and N. Cooprider, "Interrupt verification via thread verification," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 9, pp. 139–150, 2007.

[20] L. Yin, W. Dong, W. Liu, and J. Wang, "On scheduling constraint abstraction for multi-threaded program verification," *IEEE Transactions on Software Engineering*, 2018.

[21] E. Mercer and M. Jones, "Model checking machine code with the gnu debugger," in *International SPIN Workshop on Model Checking of Software*. Springer, 2005, pp. 251–265.

[22] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.

[23] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Oct. 2003.

[24] X. Wu, L. Chen, A. Miné, W. Dong, and J. Wang, "Static analysis of runtime errors in interrupt-driven programs via sequentialization," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 4, pp. 1–26, 2016.

[25] N. Chopra, R. Pai, and D. D'Souza, "Data races and static analysis for interrupt-driven kernels," in *European Symposium on Programming*. Springer, 2019, pp. 697–723.

[26] M. Pan, S. Chen, Y. Pei, T. Zhang, and X. Li, "Easy modelling and verification of unpredictable and preemptive interrupt-driven systems," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 212–222.