Static Data Race Detection for Interrupt-driven Embedded Software

Rui Chen[1] Xiangying Guo[1] Yonghao Duan[1] Bin Gu[1] Mengfei Yang[2]
[1]Beijing Institute of Control Engineering
Beijing, China
chenruis@gmail.com
[2]China Academy of Space Technology
Beijing, China
yangmf@bice.org.cn

***Abstract*—Interrupt mechanisms are widely used to process multiple concurrent tasks in the software without OS abstraction layer in various cyber physical systems (CPSs), such as space flight control systems. Data races caused by interrupt preemption frequently occur in those systems, leading to unexpected results or even severe system failures. In recent Chinese space projects, many software defects related to data races have been reported. How to detect interrupt based data races is an important issue in the quality assurance for aerospace software. In this paper, we propose a tool named RaceChecker that can statically detect data races for interrupt-driven software. Given the source code or binary code of interrupt-driven software, the tool aggressively infers information such as interrupts priority states, interrupt enable states and memory accesses at each program point using our extended interprocedural data flow analysis. With the information above, it identifies the suspicious program points that may lead to data races. RaceChecker is explicitly designed to find data race bugs in real-life aerospace software. Up to now, the tool has been applied in aerospace software V&V and found several severe data race bugs that may lead to system failures.**

CPS; embebdded software; interrupt-driven software; data race; program analysis

I. INTRODUCTION

Data races occur when multiple concurrent execution units (such as threads, tasks and interrupts) are about to access the same piece of memory, and at least one of those accesses is a write operation. Because the order of such two data access operation is undetermined, data races may lead to unexceptional software behaviors or even severe system failures. Many industry accidents are caused by data races in embedded software, such as the Therac-25 medical accelerator accident [1] which killed five lives, and the 2003 blackout in the USA and Canada [2]. Failures caused by data races always occur in special external environment and concurrent execution interleaving case, and are hard to reproduce and detect.

Due to the importance of the issue above, many researches have been conducted on the techniques and tools for detecting data race. However, most race detection methods in literature focus on multi-threaded program, which use thread mechanism to deal with concurrent tasks. Interrupt-driven software process multiple concurrent tasks based on interrupt, different on scheduling, synchronization and preemption from the thread-base programs. [3] shows that, 6.8 billion microcontroller units were shipped in 2004. Most of software built on MCUs is interrupt-driven. They are widely adopted in many fields, such as the safety critical systems in avionics, aerospace and auto electronic. According to the statistics of [4], many software bugs found in spacecraft AIT (Assembly, Integration and Test) are related to data races and interrupts.

In this paper, to capture the semantic features of interrupt based data races, we give our definition which is a little different from traditional data race definition in multithreaded programming. Bit-level-precision data flow analysis is used to track variable/register/memory values to infer interrupt preemption relations. Based on this, we implement a data race detection tool named RaceChecker using static program analysis techniques. Given the source code of an interrupt-driven program, RaceChecker aggressively infers information such as interrupts priority states, interrupt enable states and memory accesses at each program point, and then identifies the suspicious program points that may lead to data races. Currently, RaceChecker can be used to analyze the source code written in C or the executable binary files. Case studies on real-life aerospace software demonstrate the effectiveness of the tool.

The paper is organized as follows. The next section describes a real-life motivation example from spacecraft software. Section 3 presents a formal definition of interrupt-based data race. Then our detailed approach of RaceChecker is presented in Section 4. Section 5 discusses our case study. Section 6 presents related work and Section 7 concludes.

II. EXAMPLE

In this section, an example is used to explain the data race problem in interrupt driven software.

Consider the simplified code snippet shown in Fig. 1. It is a typical interrupt driven program, which has a main program and several interrupt service routines (ISR). The function `main` does several jobs in an infinite loop with a `while(1)` structure, waiting for external or internal interrupts to handle other real-time events. The function `isr1` and the function `isr2` are ISRs. The execution pointer of CPU will jump to the entry of the corresponding ISR when an interrupt is triggered, where an interrupt-style concurrent happens.

In the example, one of the most important tasks in the main loop is to compute the variable `StarTime` representing star time by reading the values of `Time.s` and `Time.ms`,

978-0-7695-4454-0/11 $26.00 © 2011 IEEE
DOI 10.1109/SSIRI-C.2011.18

representing the second and millisecond values of the star time, respectively. The computation is finished by three statements. The function `isr2`, which is a timer ISR, updates the values of `Time.s` and `Time.ms` every 20 millisecond. If a timer interrupt occurs and `isr2` is called right after line 5 is executed, `StarTime` in line 7 may be set to a wrong value, i.e., it may be computed with a fresh millisecond value and an outdated second value. To prevent such a race condition, the computation process in the function `main` need to be uninterruptable. As shown in line 4 and line 8, the program is made with attention for the condition, by disabling the timer interrupt before the computation and enabling it after the computation, which is a common-used programming strategy in interrupt driven software development. Everything looks good now.

Unfortunately, there exists another interrupt that may break the piece. In certain cases, the interrupt with respect to `isr1` may occur during the execution from line 4 to line 8, which can enable the timer interrupt by setting `ET0` to 1. If a timer interrupt enters when `isr1` returns to the interrupt point in the function main, a race condition occurs.

The example comes from real-life space software. The bug was fortunately found during the testing process, and a great loss is prevented.

In this paper, our method is designed to detect this kind of race condition, as early as possible.

```
1.  int main(void){
2.    while(1){
3.      …
4.       ET0 = 0;
5.       f1 = Time.s
6.       f2 = Time.ms;
7.       StarTime = f1 + f2*0.001;
8.       ET0 = 1;
9.       …
10.   }
11. }

12. void isr1(void){
13.    …
14.    ET0 = 0;
15.    …
16.    ET1 = 1;
17.    …
18. }

19. void isr2(void){
20.   …
21.   Time.s = newS;
22.   Time.ms = newMs;
23.   …
24. }
```

Figure 1. Sample source code that potentially causes race condition

III. BASIC DEFINITION

Before presenting the details of the method, we first give several basic definitions to describe data races for interrupt driven software.

Definition 1(*pseudo-thread*)

Pseudo-thread represents an execution unit which occupies the CPU. If the Pseudo-thread scheduler gives the CPU to program point p, any program point that can be reached from p by normal execution, jump instruction and call instruction belongs to pseudo-thread T whose entry point is p. Let *Prio(T)* be the priority of pseudo-thread T. There is a switch from T_1 to T_2 ($T_1 \rightarrow T_2$) iff $Prio(T_2) > Prio(T_1)$.

The preemption relation of pseudo-thread is transitive, e.g., if $T_1 \rightarrow T_2$, and $T_2 \rightarrow T_3$, then $T_1 \rightarrow T_3$.

For example, the main routine and every interrupt service routine are all pseudo-threads. The priority of interrupt corresponds to the pseudo-thread priority, where the main routine has a lowest priority and the interrupt priority register determines the interrupt priority.

Definition 2(*pseudo-thread enable condition*)

Denote the enable condition of pseudo-thread T by E_T. Pseudo-thread preemption occurs only if the current program state satisfies the pseudo-thread enable condition. Given pseudo-threads T_1, T_2 and program point $p \in T_1$, $T_1 \rightarrow T_2$ occurs at p only if $Prio(T_2) > Prio(T_1)$ and $E_{T2}(p)$, denoted as $T_1 \overset{P}{\rightarrow} T_2$.

Definition 3(*Data access*)

Given a program point p, let rd_p be the set of data read at p, and wt_p be the set of data written at p.

Given a pseudo-thread T, let rd_T be the set of data read in T, and wt_p be the set of data written in T.

Definition 4 (*Data race*)

Given a program point $p \in T_1$, denote the race data set by $\mathbb{R}$,

$$\mathbb{R} = rd_p \cup wt_p \cap \bigcup_{t \in \mathbb{D}} (wt_t \cup rd_t) - (rd_p \cap \bigcup_{t \in \mathbb{D}} rd_t)$$

where $\mathbb{D} = \{T \mid T_1 \overset{P}{\rightarrow} T\}$.

In other words, an interrupt data race happens when two pseudo-threads access the same shared memory location concurrently, and at least one of the two accesses is write.

The four definitions above are the basics of our method and tool.

IV. RACECHECKER

RaceChecker is a static data race detection tool for interrupt driven software. Until now, we have built two versions of RaceChecker, one for C programs, the other for binary programs. The initial goal of RaceChecker is to create an easy to use, automatic analysis tool that can assist the manual code inspection process in aerospace software V&V.

To explain the approach we take Intel MCS-51 program for example. The Intel MCS-51 is a Harvard architecture, single chip microcontroller series, which is widely used in various embedded systems. It provides many functions (CPU, RAM, ROM, I/O, interrupt logic, timer, etc.) in a single package. There are 5 interrupt sources provided with two-level priorities: 2 external interrupts, 2 timer interrupts, and the serial port

interrupt. Each of the interrupt sources can be individually enabled or disabled by setting or clearing a bit in the register named IE (Interrupt Enable). The register also contains a global disable bit, which can be cleared to disable all interrupts at once. Each interrupt source can also be individually programmed to one of two priority levels by setting or clearing a bit in the register named IP (Interrupt Priority). The main program can be interrupted by every enabled interrupt. A low-priority interrupt can be interrupted by a high interrupt, but not another low-priority.

The race detection process for MCS-51 program in RaceChecker works as follows. First, the C code or the binary code is transformed into their intermediate representations. Then, the data flow analysis is started for each pseudo-thread in the program on three abstract domains, including interrupt enable flag, interrupt priority, and indirect memory access. All of the three analysis steps are based on our extended interprocedural data flow analysis framework for interrupt driven software. After the data flow analysis, pseudo-thread enable condition and the data access can be determined for each program point. Finally, according to *Definition 4*, we visit every program point and identify the potential data races.

The rest of this section gives every step of our approach in more detail.

A. Building Intermediate Representation

In this step, there is a little difference between C code and binary code. We lex and parse the C code to abstract syntax tree (AST), build control flow graph (CFG), and create the function call relationships. For the binary code, we disassemble them to assembly code first and then built a naïve control flow graph. Note that, due to the indirect jump instructions, the naïve CFG must be reconstructed together with data flow analysis later.

In addition, in order to improve the usability of the tool, we must be able to map the analysis result of binary code to its source code. We designed a disassembly file format with line directive in to describe the map information. It is just like the output file of GCC Preprocessor. We disassemble the executables and parse the debug files at the same time to generate an intermediate file in the format mentioned above.

B. The Extended Data Flow Analysis Framework

The basic framework of our data flow analysis is derived from KilDall's iterative fixed point algorithm [5]. To improve the precision, interprocedural analysis is introduced. However, for interrupt driven software, there are the following two main challenges to meet:

1) Due to indirect jump instruction, the CFG built by syntax parsing is not complete, especially for binary code. On one hand, as we know, the data flow analysis requires CFG to propagate data flow facts. On the other hand, the indirect jump target value analysis must be finished first before rebuilding a more complete CFG. Each of the two tasks depends on the other one.

2) Interrupt preemption approximation can be determined by data flow analysis for interrupt enable register. Then it will affect program's dynamic control flow, which falls into the same issue presented above. Although interrupt handler can be seen as a special function from certain view, interrupt preemption usually makes data flow facts being propagated on an extra concurrent path.

Therefore, we proposed our extended data flow analysis approach. To deal with indirect jump instruction, the data flow analysis must be done iteratively together with CFG reconstruction. Different from traditional data flow analysis, the fixed point is reached only if the CFG and data flow facts are both unchanged. For the interrupt preemption issue, a similar strategy is used. The data flow facts are propagated from all locations where interrupts are enabled into each ISR. In this case, the fixed point is reached only if the interrupt preemption approximation and data flow facts are both unchanged for all locations.

In order to handle recursive function calls and interrupt preemptions, we develop a context sensitive interprocedural analysis approach which is extended from algorithms in [6][7]. In this algorithm, summaries are used to cache the effects of function calls and ISRs. The summary of a function or an ISR are described as a set, with each element denoting a pair of data flow facts, corresponding to the function's entry point and exit point respectively. Every time before a callee function is analyzed, the function summary is applied to avoid duplicate analysis. And every time after a callee function is analyzed, its function summary is updated immediately. The algorithm consists of the following two cases:

Case 1: The current CFG node calls a function *g*. Denote current program state by *is:*

1) Find the CFG of function *g*, and analyze it if exists.

2) Apply function *g*'s summary cache.

a) For data flow facts that hit the cache, exclude them from the current program state, and get the analysis result from the cache directly.

b) Continue to analyze function *g* with new current program state.

c) If the whole program state hit the cache, the function *g* is skipped.

d) Compute the output program state *os* of function *g* by combining the results of *(a)* and *(b)*.

3) Update Function *g*'s summary according to *is* and *os*.

4) The program state of the return site of function *g* is updated to *os*.

Case 2: For every CFG node where interrupts are enabled, analyze each ISR as a callee function. The only difference between normal function call and interrupt call is in Step *4)*. Because of the nondeterminism of the interrupt preemption, the computation of the return site's program state should be a combination, instead of a straight propagation. The Step *4)* in interrupt version is as follow:

4) The program state of the return site of interrupt is updated to $is \cup os$.

C. Interrupt Enable Analysis

The goal of Interrupt Enable Analysis (IEA) is to compute all the possible values of interrupt enable register for every program point. IEA is built on our extended data flow analysis framework, with the set of register IE value as the abstract domain. As mentioned earlier in this section, each bit of IE has its own meaning, so the abstract domain of interrupt enable analysis must be bit-level precise.

To avoid the state explosion problem, we adopt a tricky state grouping strategy shown in Fig. 2. Each time the IE state set needs to be updated, the procedure *combine* is called. In Fig. 2, we use symbol *e.8* to denote the 8th bit of IE, the global interrupt enable bit.

```
IEState = φ
procedure combine(newIE)
begin
 if ∃e∈IEState(e.8=newIE.8) then
    e := e | newIE
 else
    IEState := IEState∪newIE
end
```

Figure 2. Grouping algorithm in IEA

For example, suppose current *IEState={0x81,0x03}*, *newIE=0x82*, then the procedure *combine* changes *IEState* to *{0x83, 0x03}*, the first element of which is the result of *0x81 BOR 0x83*. That is, *IEState* always contains at most two elements, each of which has different global interrupt bit state.

Using the grouping algorithm, the size of program states maintained in data flow analysis can be linear with respect to the program size. At the same time, compared to set combination operation, no loss of precision is introduced.

D. Interrupt Priority Analysis

Interrupt Priority Analysis (IPA) is to compute all the possible values of register IP for all program locations. The method used in IPA is similar as IEA. The only difference is that, the abstract domain of IPA is the value of IP, instead of IE.

E. Memory Access Analysis

We use Memory Access Analysis (MAA) to infer memory read/write sets for all program locations. For binary code and C code, the procedure is different:

Binary code. Memory access for indirect addressing instruction, i.e. `MOV @DPTR, A`, could not be determined simply. To solve this problem, the value analysis of indirect addressing register must be done first.

C code. Instead of indirect addressing, pointers lead to the similar problem for source code analysis. In our current implementation, Steensgaard's algorithm is used for simple pointer analysis. Fortunately, in interrupt driven software, the pointer usage is not complicated, and our pointer analysis could always infer most required information.

F. Identifying data races

In this step, every program point p for every pseudo-thread is visited to identify all potential data races. It consists of the following 4 sub steps:

1) Determine interrupt set $\mathbb{I}$, that can interrupt the current pseudo-thread at p, using the results of IEA and IPA.

2) For each interrupt $i \in \mathbb{I}$, compute the intersection $\mathbb{R}$ for memory access set of p and memory access set of i.

3) Refine $\mathbb{R}$. Each element of $\mathbb{R}$ corresponds to two memory accesses, one of which comes from p, and the other from i. Exclude element e from $\mathbb{R}$ unless at least one of the two accesses is write.

4) Finally, we get a refined $\mathbb{R}'$ as data race detection result for p.

Note that, the data races reported by RaceChecker can be divided into three categories: harmful races, intentional races, and false positives. To confirm a real bug, users have to inspect the report manually.

V. Application of RaceChecker

This section describes case studies on applying RaceChecker to find data race bugs for real life aerospace software. The aim of this study is to demonstrate the effectiveness of our method.

We select three onboard software as the detection target of our tool:

1) ***SSCS***. The main function of this software is to drive the solar sailboard on demand. *SSCS* affects energy supply of the satellite. The software is programmed in C.

2) ***RTU***. *RTU* runs on a remote terminal unit of satellite. It communicates with the central computer of satellite (CTU), processing/transmitting data, and executing indirect command from CTU additionally. The software is programmed in assembly language originally.

3) ***BCS***. *BCS* controls beaconing on a communication satellite. It is programmed in assembly language.

All of the three embedded software run on MCS-51 CPUs, and are in small size. RaceChecker analyzes each of them in less than 60 seconds.

For each analyzed software, RaceChecker reported tens of data races. We invited authors and testers of the software to validate these results. Finally, 1, 2, 1 bugs are confirmed for *SSCS*, *RTU* and *BCS* respectively, as shown in Tab. 1.

The bug found in *SSCS* is mission critical. The example described in Section 2 origins from this bug. The bug leads to a wrong angle value which might make the satellite in short energy supply.

The other bugs found in *RTU* and *BCS* fall into one category. They happen when only one of two related variable is modified in interrupt handler. The type of unexpected modification leads to inconsistency, which usually brings in functionality fault.

TABLE I. APPLICATION RESULT OF RACECHECKER

Software	*Source Type*	*Size(KLOC)*	*Bugs Found*
SSCS	C	2.10	1
RTU	Binary code	4.47	2
BCS	C	1.26	1

Until now, RaceChecker has been applied to checking embedded software of Chinese Lunar Exploration Rover, ShenZhou Manned Spaceship, and other spacecraft systems, all of which are typical CPSs.

VI. RELATED WORK

In literatures, two ways can be used to reduce the occurrence of data race in applications: finding data racing in software testing or verification, and prevent data race by programming language. In this paper, we focus on the first way, i.e., data race detection. Lots of data race detection methods have been proposed. This section presents them from the following three aspects.

A. Static Methods

Static data race detection methods do not require the execution of the program, and can be divided into three classes: type-based, flow-based and model checking-based.

Type-based approaches detect data races using type checking techniques. Generally speaking, most type-based tools required source code annotations to specify synchronization operations [8][9]. However, Due to great manual labor work to annotate, type-based methods can not been applied in practice. Therefore, some methods [10][11] are proposed recently to infer annotation information automatically.

The flow-based detectors involved data flow analysis or control flow analysis techniques [12][13], which show their effectiveness by outstanding experimental results. Most of them adopt static lockset algorithms, and can scale to large programs in MLOC size. However, due to the undecidability of program behavior, all of the static tools suffer from high false positives rate.

Besides, most of the flow-based tools assume lock based synchronization in their approach. For different programming schema, the protection and synchronization for shared data could be achieved by different strategy, i.e. semaphore, where the methods above may not deal with.

Researchers also proposed some model checking based approaches [14][15], which search reachable race condition by traverse all possible thread interleaving. Model checkers are always path sensitive and not specific to some synchronization operation. They can give counter examples when finding violations to refine the abstraction with higher precision. However, as we known, model checking suffers from the state explosion problem, which made the application to large program more difficult.

B. Dynamic Methods

Dynamic data race detection methods require the execution of the program. Then certain algorithms will be used to find the suspicious code that contains data race from the collected execution data. Lots of dynamic methods have been proposed since 1970's. For example, Lamport [16] proposed happens-before relation which is still the basic idea of many dynamic tools nowadays. Happens-before based tools is precise with no false positives but with many false negatives. The lockset algorithm is then proposed to reduce false negatives originally [17], which is tailored to the common lock-based synchronization discipline. The primary problems with lockset-based dynamic approaches are that it produces many false positives as well as having the usual potential for false negatives of any dynamic analysis.

In order to collect the execution trace information, dynamic tools required the program to be instrumented first, involving high cost. Recent researches focused on how to reduce the cost and how to improve the precision of race detection. Choi et al. [18] proposed a novel approach that achieved very few false positives and low runtime overhead with a combination of complementary static and dynamic optimization techniques. Marino et al. [19] proposed a happens-before based method by sampling execution traces to reduce the overhead. Yu et al. [20] adopt an adaptive strategy that can direct more effort to areas that are more suspicious. Lu et al. proposed to improve the efficiency of data race using hardware tools [21].

Due to the usage of execution data, dynamic data race detection methods may be more precise than the static methods. However, this kind of methods depends on the sufficient executions and test input. And for the programs whose execution costs are high, such as the embedded software used on embedded systems, these methods are unsuitable.

C. Race Detection Related to Interrupt

All of the methods above are used in data race detection for multi-thread programs. In literatures, few researches aim at the data race detection for the interrupt driven embedded programs. The exceptions include the follows. Regehr et al. [22] proposed a novel way to verify the correctness of the concurrency in interrupt driven programs. They proposed to convert the interrupt driven program into a thread-based program, which can be checked by an existing thread checking tool. However, as the authors claimed, this method is not rigorous and should be formalized in future work. Mercer and Jones [23] exploit GDB's state saving and restoring capabilities to model check interrupt driven embedded code. The SLAM model checker [24] and the RacerX [12] race detector are used to find bugs in kernel code, including interrupt handler.

VII. CONCLUSION

Motivated by a category of real-life software bugs that Chinese aerospace software suffered from, we have proposed a static data race detection approach in this paper. To validate our approach, we implemented a tool named RaceChecker, and applied it to real-life embedded software of aerospace system. The case studies show that, RaceChecker can find data race bugs in real-life aerospace software efficiently.

Future works will focus on eliminating false positives, identifying harmful races, and refining the report of static analysis with dynamic analysis.

ACKNOWLEDGMENT

We would like to thank Lei Zhao for his insightful suggestion on this work. We are grateful to the anonymous reviewers for their helpful comments on a preliminary version of this paper. This work is funded by National Science Foundation of China (Grant No. 91018014 and No. 90818024). This work is also supported by Independent Innovation Program of Beijing Institute of Control Engineering.

REFERENCES

[1] N.G. Leveson, C. Turner. An investigation of the Therac-25 accidents. IEEE Computer, July 1993, pages 18-41.

[2] US-Canada Power System Outage Task Force. Final report on the August 14, 2003 blackout in the United States and Canada: Causes and Recommendations. North American Electric Reliability Council, April 15, 2004.

[3] M. Baronand and C. Cadden. Strong growth to continue for MCU market, http://www.instat.com/press.asp?ID=1445&sku=IN0502457SI, 2005.

[4] Flight software assurance center of China Academy of Space Technology. Quality report of Chinese satellite on-board software, 2009.

[5] G. A. Kildall. A unified approach to global program optimization. In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 194–206, 1973.

[6] Xie Yichen, Aiken Alexander. Saturn: A SAT-Based Tool for Bug Detection[C]. Proceedings of CAV,2005: 139-143

[7] Engler D., Chelf B., et al. Checking System Rules Using System-specific Programmer-written Compiler Extensions[C]. Proceedings of OSDI, 2000:1-1

[8] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2001.

[9] D. Grossman. Type-Safe Multithreading in Cyclone. In Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI), pages13-25,2003.

[10] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), pages 90-96, 2001.

[11] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), pages 83-94, 2005.

[12] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In ACM Symposium on Operating Systems Principles (SOSP), 2003.

[13] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE), pages 205-214, 2007.

[14] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In Proceedings of ACM SIGPLAN 2004 conference on Programming Language Design and Implementation (PLDI'04), pages 14-24, 2004.

[15] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. SIGPLAN Notice, 39(6):1–13, 2004.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of ACM, 21(7):558–565, 1978.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T.E. Anderson, Eraser: A dynamic data race detector for multi-threaded programs, ACM Transactions on Computer Systems 15 (4):391–411, 1997.

[18] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02), pages 25-269, 2002.

[19] D. Marino, M. Musuvathi and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09), 2009.

[20] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In Proceedings of the twentieth ACM Symposium on Operating Systems Principles (SOSP '05), pages 221-234, 2005.

[21] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[22] J. Regehr, N. Cooprider. Interrupt verification via thread verification. Electron. Notes Theoretical Computer Science 174(9), 2007.

[23] E. G. Mercer, M. D. Jones. Model checking machine code with the GNU debugger. In Proceedings of the SPIN Workshop on Model Checking of Software (SPIN'05), San Francisco, 2005.

[24] T. Ball, E. Bounimova, and etc. Thorough static analysis of device drivers. In Proceedings of the 1st EuroSys Conference, Leuven, Belgium, 2006.