Efficient Data-Race Detection with Dynamic Symbolic Execution

Andreas Ibing
Chair for IT Security, TU München
Boltzmannstrasse 3, 85748 Garching, Germany

***Abstract*—This paper presents data race detection using dynamic symbolic execution and hybrid lockset / happens-before analysis. Symbolic execution is used to explore the execution tree of multi-threaded software for FIFO scheduling on a single CPU core. Compared to exploring the joint scheduling and execution tree, the combinatorial explosion is drastically reduced. An SMT solver is used to control a debugger's machine interface for adaptive dynamic instrumentation to drive program execution into desired paths. Data races are detected in concrete execution with available static binary instrumentation using hybrid analysis. State interpolation using unsatisfiable cores is employed for path pruning, to avoid exploration of paths that do not contribute to increasing branch coverage. An implementation in Eclipse CDT is described and evaluated with data race test cases from the Juliet C/C++ test suite for program analyzers.**

***Index Terms*—race detection; symbolic execution; interpolation; branch coverage.**

I. Introduction

A data race means, that there are concurrent accesses from different threads to the same variable, of which at least one is a write. Data race bugs are introduced in multi-threaded software when the developer forgets to lock a resource, that is shared between threads. Because races are observed only for certain thread interleavings depending on the scheduler's decisions, they are difficult to reproduce and sometimes called 'Heisenbugs'. Exactly locating feasible data races is known to be NP hard [1].

Data race detection is typically implemented as dynamic analysis with binary instrumentation. It follows happens-before analysis with vector clocks [2], or lockset analysis [3], or a hybrid algorithm [4]. These algorithms introduce some false positive and / or false negative detections. A prominent example is ThreadSanitizer [5], which is integrated with the GNU and Clang C compilers. It is reported to slow down execution speed by at least a factor of ten. Therefore, it is typically applied as dynamic analysis with a manually written test-suite. The achievable code coverage is then limited to execution coverage of the test suite. Therefore, data race detection in this approach also depends on the size of the available test suite, which is limited by cost constraints.

Symbolic execution [6] automatically performs a systematic program path exploration and enables program coverage independent of a test suite. Program input is treated as symbolic variables, and operations on the variables are translated into logic equations. The feasibility of program paths (feasible with any program input) is then decided with a Satisfiability Modulo Theories (SMT, [7]) solver. Symbolic execution is often applied as dynamic analysis in the form of 'concolic' (concrete and symbolic) execution [8], [9]. The program is executed with concrete input, while symbolic constraints are collected on the program path. The input for the next path is generated by the SMT solver, so that the program takes the desired path. Concolic execution offers the possibility for consistent concretization of formulas (fallback to concrete value). This is useful if certain constructs can not be handled with the solver. Concretization does not introduce false positive path satisfiability decisions or error detections, but in general introduces false negatives.

The prominent symbolic execution tools DART [8], CUTE [9] and KLEE [10] currently do not feature data race detection. Symbolic execution tools that do support race detection are jCUTE [11], Con2colic [12] and LCT [13]. They use a solver to search paths through the program's joint execution and scheduling tree, i.e., the solver determines both program input and thread scheduling. The combinatorial explosion can be partly mitigated with partial order reduction [14]. The resulting race detection with symbolic execution is considerably more complex and slower compared to symbolic execution of single-threaded code.

This paper presents data race detection using dynamic symbolic execution and hybrid lockset / happens-before analysis. Symbolic execution is used to explore the execution tree of multi-threaded software for FIFO scheduling on a single CPU core. Complexity and scaling of symbolic execution are improved by interpolation based path pruning.

The remainder of this paper is organized as follows: Section II motivates and describes the algorithm, Section III depicts its implementation. In Section IV, the implementation is evaluated with data race test cases from the Juliet test suite [15]. Related work is reviewed in Section V. Results of the experiments are discussed in Section VI.

II. Algorithm

A. Motivation

The algorithm is motivated by the following aspects:

- The operating system scheduler can be used in concolic execution for speed-up. Code execution is faster than interpretation. Symbolic execution needs a reproducible execution tree, also for multi-threaded software. This can be achieved by restricting the scheduling to one CPU core

and to reproducible scheduling independent of system load.
- Race detection works faster during concrete execution. Event tracing and instrumentation do not need a symbolic interpreter.
- The analysis of a program path should continue after the detection of a potential data race, in order to detect further errors along path extensions. This means, that actual races should be avoided while still detecting them. Program behaviour without races is independent of scheduling. Data races can be prevented by using FIFO scheduling on one CPU core. With this scheduling, potential data races can still be detected (using happens-before, lockset or hybrid analysis).
- State interpolation can be used to improve scaling of symbolic execution. Unsatisfiable branches can be interpolated by computing unsatisfiable equation cores [16]. The interpolation can be used to prune paths, that are redundant with respect to coverage. Unsatisfiable cores can be backtracked by approximate weakest-precondition computation. This requires depth-first path exploration.

An advantage of this approach is that bugs other than races (e.g., buffer overflows) can be found just like with symbolic execution of single-threaded code. Only one representative thread interleaving is analyzed per executed program path. With FIFO scheduling on one CPU core, there is also the possibility to apply standard code coverage criteria like in single-threaded execution.

B. Dynamic Symbolic Execution

Code execution is faster than interpretation, and especially faster than translation into logic formulas. Therefore, only as few code locations as needed are interpreted symbolically, otherwise the code is executed concretely. These locations are the definition of input-dependent variables (symbolic variables) and input-dependent branch decisions (dependent on a symbolic variables). Because it is context-sensitive which variables are symbolic, adaptive dynamic instrumentation is used. The program under analysis is executed concretely. If a program location needs formula generation, the analysis switches to symbolic interpretation and generates the constraint formula. Then, further dependent locations are marked for symbolic interpretation, and the concrete execution is continued.

1) Reproducible Execution Tree for Multi-Threaded Programs: Multi-threaded software can be described as state transistion system with a combined scheduling and execution tree. Considering a deterministic scheduling algorithm without outside parameters (independent of system load etc.), the execution tree for this scheduling is yielded. Here, FIFO scheduling on one CPU core is used. This avoids data races because only one thread at a time is active, and it is not preempted. Computations in a thread between calls to the scheduler become atomic. Data race detection is still possible. The execution further becomes reproducible: when restarted with the same program input, the identical thread interleaving is yielded.

2) Execution Tree Exploration: Symbolic program input is configurable. It can comprise command line parameters and system call return values. In the execution of the first program path, pre-configured standard return values (that are always valid) from the symbolic system calls are used for concrete execution. Then, the solver is used to generate concrete input values for the next path from the collected constraints of the previous path. The last symbolic branch condition is negated, if the negation is not yet covered in this context. If the resulting equation system is unsatisfiable, the branch decision is backtracked. This path exploration is depth-first search. Without state interpolation and path pruning, it explores all satisfiable program paths.

3) Interpolation Based Path Pruning: The state interpolation uses unsatisfiable core (unsat-core) computation with serial constraint elimination as described in [16]. Given an unsatisfiable conjunction of formulas, an unsat-core is a subset of the formulas whose conjunction is still unsatisfiable. If input generation for a path is unsatisfiable, an unsat-core is computed from the path constraint. Unsat-cores are backtracked during depth-first path exploration. A constraint is removed from the unsat-core when the control flow graph (CFG) node is backtracked, for which the constraint was generated.

a) Pruning: Prune formulas are the backtracked unsat-cores. When a control flow decision node is backtracked, the conjunction of the branch prune formulas is used. Branch targets are used as potential prune points. When symbolic execution reaches a branch for that a prune formula has been computed, then the solver is used to decide whether the current path is redundant and can be pruned. If the current path's path constraint implies the prune formula, then the path is pruned. This approach can prune paths from different contexts [16]. The implication means that all branches, that were unsatisfiable in the previous context, are also unsatisfiable in the current context. Extensions of the path would therefore not contribute to increasing branch coverage [17].

C. Dynamic Race Detection

Data races are detected with hybrid dynamic analysis during concrete execution, using the ThreadSanitizer algorithm [5]. This subsection shortly reviews the algorithm. Happens-before analysis with vector clocks is combined with lockset analysis to reduce the number of false negative detections. The hybrid detection has false positive detections. False positives can be eliminated by adding annotations to the program. Binary instrumentation is used to trace the relevant events [5]:

- memory access events: read and write
- synchronization events: locking and happens-before arcs. Locking events are write lock, read lock, write unlock and read unlock. Happens-before events are signal and wait.

The events are traced as state machines in shadow memory. The state machines can be run as pure happens-before or as hybrid analysis, with configurable context tracing information (speed versus information).

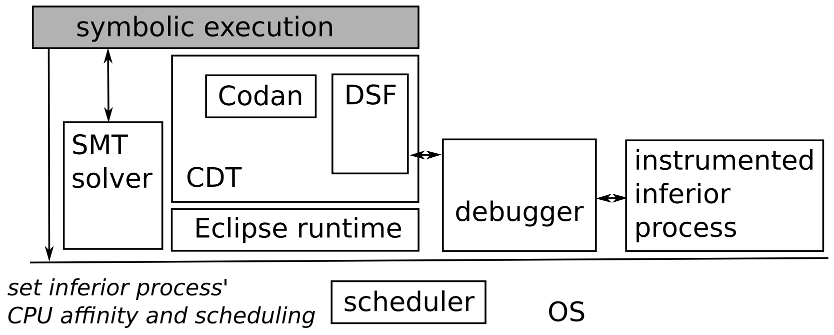

symbolic execution
Codan
DSF
SMT solver
CDT
Eclipse runtime
debugger
instrumented inferior process
set inferior process' CPU affinity and scheduling
scheduler
OS

Fig. 1. Overview

III. Implementation

An overview of the main components is shown in Figure 1. These comprise the Eclipse runtime with plug-ins for the C/C++ development tools (CDT). CDT contains a code analysis framework (Codan) and debugger services framework (DSF). The instrumented program under test is controlled through DSF using a debugger (here the GNU debugger `gdb`). The debug inferior process is scheduled by the operating system scheduler. Symbolic execution is implemented as Eclipse plug-in on top of CDT. Logic formulas are decided using the Z3 SMT solver [18].

A. Debugger based concolic execution of multi-threaded code

The implementation extends the dynamic symbolic execution engine for single-threaded code presented in [17]. It uses the C/C++ parser from Eclipse CDT and the control flow graph builder from Codan. The debugger services framework is an abstraction layer over debuggers' machine interfaces. The program under test is executed in a debugger.

a) Selective symbolic interpretation: Only when the debugger hits a breakpoint, then a constraint is generated for the respective code location. The breakpoints are updated so that the debugger breaks on usage and definition of symbolic variables, then the debugger continues. At a breakpoint, the CFG node is resolved for the thread that stopped. Tree-based translation is used to generate the logic constraint. The debugger is queried for values of concrete variables where needed. Variables can become symbolic (by assignment of a symbolic value) and concrete (by assignment of a concrete value).

b) Static pre-analysis: Initial breakpoint locations are determined with static analysis before starting the symbolic execution. These locations are the definition of symbolic program input and input-dependent branches (branch locations that are input-dependent on any branch). In addition, the analysis overapproximates the set of pointers that might on any program path point to a symbolic target. If a target becomes symbolic during symbolic execution, a breakpoint is set on usage and definition of all pointers, that might point to this target. The static pre-analysis could be called 'maybe symbolic' analysis.

c) Single-core FIFO scheduling: The implementation currently runs on Linux, which supports different scheduling algorithms at the same time for different processes. Differing from the standard scheduler `SCHED_OTHER`, for the program under test the FIFO scheduler (`SCHED_FIFO`) is used. The CPU affinity is restricted to one CPU core. The corresponding Linux commands are `chrt` (to set the scheduling) and `taskset` (for CPU affinity).

d) Translation into SMT logic: The tree-based translation is implemented with CDT's abstract syntax tree (AST) visitor class. A control flow graph node references an AST subtree, that is traversed to generate a logic constraint. The translation uses bit-vectors and arrays. The solver is neither aware of multiple threads, nor of any scheduling. It just gets a conjunction of constraints that were collected along the program path.

e) Backtracking unsat-cores and path pruning: Unsat-cores are computed with serial constraint deletion as in [16]. There is one pass through the collected constraints on the path beginning from program start. Each constraint is tested once: if it can be removed and the rest remains unsatisfiable, then it is removed. Backtracking considers only CFG nodes that have been interpreted (where the debugger stopped). If a CFG node is backtracked, then any constraint generated for this node is removed. When a decision node is backtracked, the conjunction of the formulas from the branches is used [16]. When backtracking reaches a branch node, a prune formula is connected with this location. The prune formula is the conjunction of backtracked unsat cores. When a branch target is reached in (forward) symbolic execution, it is checked, whether there is a prune formula. If yes, then it is checked with the solver whether the path constraint implies the prune formula (i.e., whether the negation of the implication is unsatisfiable). In this case, the path can not contribute to increase branch coverage, and therefore is pruned.

B. Race detection in concrete execution using ThreadSanitizer

The implementation uses compiler instrumentation with `ThreadSanitizer` [5], which is featured by the GNU C compiler. The program under test is linked statically with the `ThreadSanitizer` library, and a breakpoint is set on the race detection error report function. If this breakpoint is hit,

```
1  #define N_ITERS 1000000
2  void CWE366_Race_Condition_Within_Thread__int_byref_12_bad() {
3    if(global_returns_t_or_f())      {
4      std_thread thread_a = NULL, thread_b = NULL;
5      int val = 0;
6      if (!std_thread_create(helper_bad, (void*)&val, &thread_a)) {
7        thread_a = NULL;
8      }
9      if (!std_thread_create(helper_bad, (void*)&val, &thread_b)) {
10       thread_b = NULL;
11     }
12     if (thread_a && std_thread_join(thread_a)) std_thread_destroy(thread_a);
13     if (thread_b && std_thread_join(thread_b)) std_thread_destroy(thread_b);
14     printIntLine(val);
15   } else {
16     std_thread thread_a = NULL, thread_b = NULL;
17     int val = 0;
18     if (!std_thread_lock_create(&g_good_lock)) { return; }
19     if (!std_thread_create(helper_good, (void*)&val, &thread_a)) {
20       thread_a = NULL;
21     }
22     if (!std_thread_create(helper_good, (void*)&val, &thread_b)) {
23       thread_b = NULL;
24     }
25     if (thread_a && std_thread_join(thread_a)) std_thread_destroy(thread_a);
26     if (thread_b && std_thread_join(thread_b)) std_thread_destroy(thread_b);
27     std_thread_lock_destroy(g_good_lock);
28     printIntLine(val);
29 } }
30 static void helper_bad(void *args) {
31   int *p_val = (int*)args;
32   for (int i = 0; i < N_ITERS; i++)      {
33     *p_val = *p_val + 1;
34 } }
35 int global_returns_t_or_f() {
36   return (rand() % 2);
37 }
```

Fig. 2. Example 'bad' function from [15] that contains a data race in line 33

the stack is traced back to a source file location, where the race is reported.

ThreadSanitizer supports dynamic annotations (C makros), which can be used if standard Posix threads are not used. They can also be used to eliminate false positive detections and to hide benign races [5]. Parts of the code can be marked as safe by the tool user. ThreadSanitizer can be run as happens-before or hybrid analysis. Also in pure happens-before mode it can report the involved locks. The slow-down by the instrumentation is reported as factor 20 – 50, and up to several hundred MB can be consumed for shadow memory [5].

IV. Experiments

a) Test cases and test setup: The implementation is evaluated with the data race test cases from the Juliet suite [15] for common weakness CWE-366 'race condition within a thread'. The test cases are 38 small artificial programs with 5-7 threads each. They contain 'good' functions (without data race) as well as 'bad' functions (that contain a data race) in order to measure false positive and false negative detections. There are two sets of 19 programs each. One set contains data races on global variables, the other contains data races on stack variables with access through pointers. Both sets cover the same 19 different data and control flow variants, that include conditional branches, loops, goto statements etc. The tests are run as JUnit plug-in tests with Eclipse 4.5.1 and gdb version 7.10 on Linux kernel 4.2.0, on a Core i7-4650U CPU. The programs under test are run as unoptimized code with default

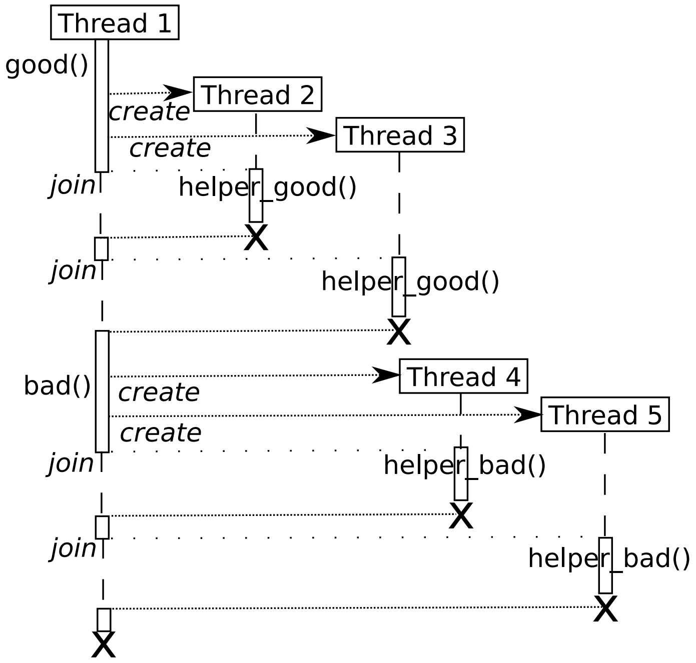


Fig. 3. Example, FIFO scheduling on one CPU core

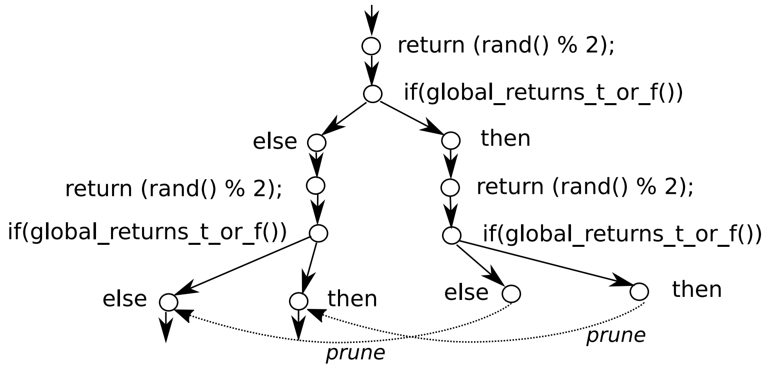


Fig. 4. Example, traversed part of execution tree

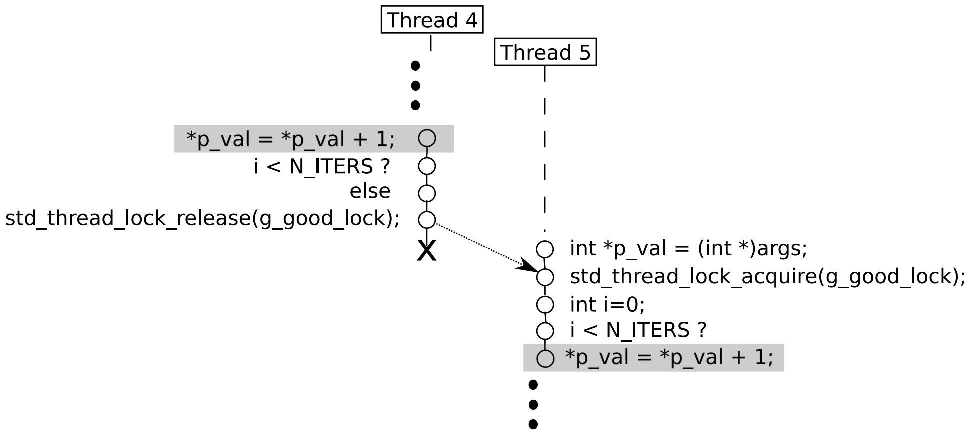

Thread 4
Thread 5
*p_val = *p_val + 1;
i < N_ITERS ?
else
std_thread_lock_release(g_good_lock);
X
int *p_val = (int *)args;
std_thread_lock_acquire(g_good_lock);
int i=0;
i < N_ITERS ?
*p_val = *p_val + 1;

Fig. 5. Happens-before analysis, no data race for example 'bad' function if the `else` branch is taken (Figure 2 line 16)

`ThreadSanitizer` settings without any annotations.

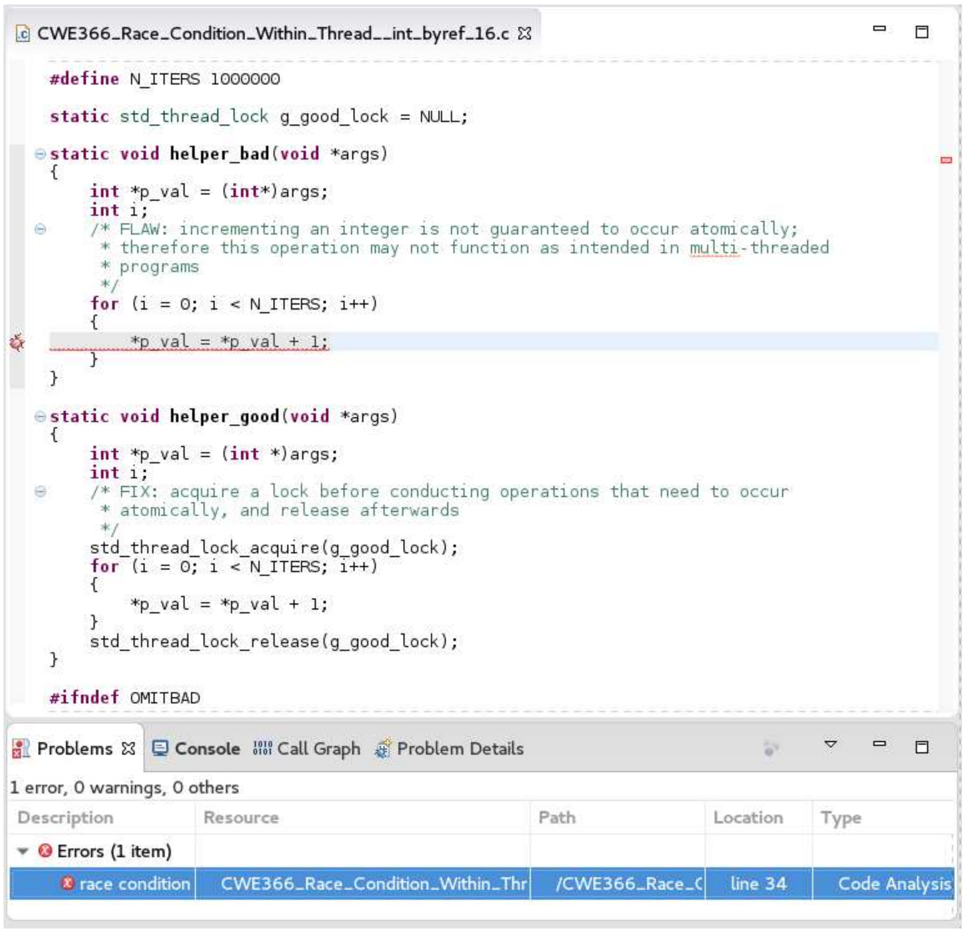
b) Example: An example 'bad' function is shown in Figure 2. It contains a data race on a stack variable with access by reference in line 33. The control flow depends on random input that is returned by a global function. The `rand()` call in line 36 generates this program input, i.e., the debugger breaks at this call, and the return value of `rand()` is treated as unconstrained symbolic variable. Before this 'bad' function, the program executes a similar 'good' function with proper locking in both branches, that also executes a loop with 10^6 iterations. The test program contains many branches, but few of them are input dependent. The loops in the 'good' and 'bad' functions are executed concretely. Because they contain only input-independent variables and branches, there is no need for symbolic interpretation. The symbolic execution finds four satisfiable program paths that non-deterministically depend on two `rand()` calls, and of which two paths exhibit the data race. FIFO scheduling on one CPU core for this program is illustrated in Figure 3. This figure looks the same for any of the four paths. The part of the execution tree (under this scheduling), that is traversed by symbolic execution, is illustrated in Figure 4. It only shows the locations where the debugger stopped. Two paths are followed to program end, the other two are pruned as indicated in the figure. The data race is accurately detected by `ThreadSanitizer`. An example part of the happens-before analysis is illustrated in Figure 5, for the `else` branch of the example function, where proper locking is used. The shaded memory access events are separated by a locking arc.

c) Results: All data races in the 38 test programs are accurately detected without false positives or false negatives. The (wall-clock) analysis runtimes are shown in Figure 7. The horizontal axis indicates the Juliet flow variant number. The average analysis runtime is below 2s. Error reporting through the Codan framework in the Eclipse GUI is shown in Figure 6.

V. Related Work

Exactly locating feasible data races is known to be NP hard [1]. The main approaches for practical race detection are usage of the happens-before relation with vector-clocks [2], or usage of locksets [3], or a combination of them. These approaches detect races when they might occur, i.e., not only with the current thread interleaving, but also with different interleavings. Main analysis methods are static analysis, model checking, dynamic analysis and symbolic execution.

a) Static Analysis: offers a possibility to detect races without false negatives. A method that requires programmer annotations and is based on type inference is described in [19]. The tools `RELAY` [20] and `LOCKSMITH` [21] apply lockset


CWE366_Race_Condition_Within_Thread__int_byref_16.c

```c
#define N_ITERS 1000000

static std_thread_lock g_good_lock = NULL;

static void helper_bad(void *args)
{
    int *p_val = (int*)args;
    int i;
    /* FLAW: incrementing an integer is not guaranteed to occur atomically;
     * therefore this operation may not function as intended in multi-threaded
     * programs
     */
    for (i = 0; i < N_ITERS; i++)
    {
        *p_val = *p_val + 1;
    }
}

static void helper_good(void *args)
{
    int *p_val = (int *)args;
    int i;
    /* FIX: acquire a lock before conducting operations that need to occur
     * atomically, and release afterwards
     */
    std_thread_lock_acquire(g_good_lock);
    for (i = 0; i < N_ITERS; i++)
    {
        *p_val = *p_val + 1;
    }
    std_thread_lock_release(g_good_lock);
}

#ifndef OMITBAD
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
race condition	CWE366_Race_Condition_Within_Thr	/CWE366_Race_C	line 34	Code Analysis

Fig. 6. Error reporting

analysis statically with data flow analysis. According to Palsberg [22], ”the best existing static technique” is implemented in `Chord` [23], but it ”reports a large number of false positives that would be daunting to examine by hand”.

b) Model Checking: In symbolic model checking, a program is translated into a logic formula, and properties are checked with an SMT solver. In theory, model checking allows for accurate race detection. It explores the symbolic state-space, that is the combined execution and scheduling tree. In practice, model checking does not scale well due to combinatorial explosion. One practical approach is bounded model checking [24], [25], [26], where the combined execution and scheduling tree is pruned with limits for the number of context switches and loop unrollings. Another way of pruning the tree is partial order reduction [27], [28], [14], [29], which prunes away irrelevant thread interleavings. Dynamic partial order reduction [14] traces the happens-before relation for thread interactions to find backtracking points for branching [14]. Optimal dynamic partial order reduction [29] explores a minimum number of representative thread interleavings.

c) Dynamic detection at runtime: is a practical way for race detection. It does not need a constraint solver. One technique is to instrument memory accesses and thread interaction with binary instrumentation and check for races using the happens-before relation [2] with vector-clocks. Happens-before analysis may have false negative detections depending on the scheduling. It is more sophisticated than lockset analysis, but scales worse with an increasing number of threads. `LiteRace` applies sampling, i.e., it monitors only a subset of all memory accesses. It can detect a majority of races by monitoring a small number of accesses [30]. `FastTrack` is an optimized implementation of happens-before analysis with reduced complexity [31]. `Pacer` [32] combines sampling with `FastTrack`. `DataCollider` [33] implements memory access sampling with hardware breakpoints and watchpoints and applies it to kernel code. Lockset analysis is used in `Eraser` [3]. It instruments memory accesses and traces thread locksets and variable locksets. If a variable access is not protected by a lock, then a warning is issued. Lockset analysis is lightweight and scales well. On the downside, it leads to more false positive detections than happens-before analysis. Hybrid race detection is presented in [4] as a two-pass solution. First, locksets are used to find problematic variables. Then, happens-before analysis is applied only to those variables. In [34], the DJIT algorithm is presented, which is a variation of happens-before analysis. `MultiRace` [35] combines DJIT with locksets to reduce false positives. A location’s lockset is reset

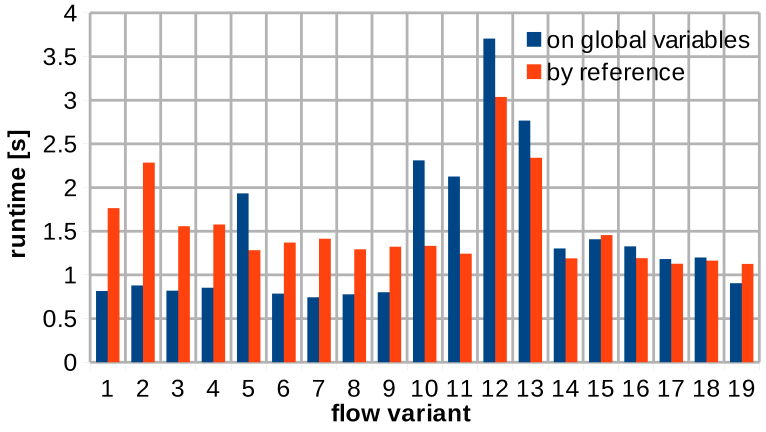


Fig. 7. Analysis runtime for data race tests from Juliet suite

at synchronization barriers. `ThreadSanitizer` [5] applies static binary instrumentation for happens-before and lockset analysis and is integrated with several current C compilers. Dynamic race detection is integrated in the managed runtime environments `RaceTrack` [36] and `Goldilocks` [37]. In [38], [39], it is proposed to integrate hardware acceleration for race detection into CPUs.

d) Symbolic execution: The prominent symbolic execution tools `DART` [8], `CUTE` [9] and `KLEE` [10] currently do not feature race detection. `jCute` [11] determines program input and thread schedule with the solver to explore different paths and interleavings, and it detects races when they occur. `Con2colic` also determines input and schedule with the solver. It implements a heuristic to first achieve branch coverage, and then explore an increasing number of context switches. Also `Con2colic` can detect a race when it occurs (no happens-before or lockset analysis). `LCT` [13] implements concolic execution with dynamic partial order reduction. In [22], the tool `Racageddon` is described. It starts with race candidates that have been found with an existing hybrid technique. It then uses concolic execution to search for input and schedule that lead to a real race (to remove false positives). `WHOOP` [40] considers races between pairs of entry points to driver code and runs a symbolic lockset analysis with SMT solver. In [16], [41], it is described that a program path can be pruned if the context implies a previously computed interpolant for the same program location. A combination of partial order reduction with interpolation based path pruning is described in [42].

e) This paper: differs from previous work on race detection with symbolic execution both in that it factors out the scheduling, and in that it applies hybrid data race detection during concrete execution. It extends own prior work [17] (on dynamic symbolic execution of single-threaded code with interpolation based path pruning) with support for multi-threaded execution and with data race detection.

VI. Discussion

Symbolic execution with FIFO scheduling on one core is used to automatically drive concrete execution into program paths of interest. The scheduling effectuates a reproducible execution tree for multi-threaded code. FIFO scheduling avoids triggering data races. Races are detected during concrete execution by instrumenting the program under test with the available `ThreadSanitizer`. The analysis is comparatively fast through concrete scheduling and concrete race detection. Path pruning is used based on interpolation of unsatisfiable branches with unsatisfiable cores. Implication checking with SMT solver assures that only paths are pruned, that can not contribute to increasing branch coverage. It is also possible to run the analysis in a virtual machine like `qemu`, that contains a `gdb` server.

Acknowledgement

This work was funded by the German Ministry for Education and Research (BMBF) under grant 01IS13020.

References

[1] R. Netzer and B. Miller, “What are race conditions?: Some issues and formalizations,” *ACM Letters on Programming Languages and Systems*, pp. 74–88, 1992. [Online]. Available: http://dx.doi.org/10.1145/130616.130623
[2] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. [Online]. Available: http://dx.doi.org/10.1145/359545.359563
[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multi-threaded programs,” *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997. [Online]. Available: http://dx.doi.org/10.1145/268998.266641
[4] R. O’Callahan and J. Choi, “Hybrid dynamic data race detection,” in *ACM Symposium on Principles and Practice of Parallel Programming*, 2003. [Online]. Available: http://dx.doi.org/10.1145/966049.781528
[5] K. Serebryany and T. Iskhodzhanov, “ThreadSanitizer: data race detection in practice,” in *Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71. [Online]. Available: http://dx.doi.org/10.1145/1791194.1791203
[6] J. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: http://dx.doi.org/10.1145/360248.360252
[7] L. de Moura and N. Bjorner, “Satisfiability modulo theories: Introduction and applications,” *Communications of the ACM*, vol. 54, no. 9, 2011. [Online]. Available: http://dx.doi.org/10.1145/1995376.1995394
[8] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Conference on Programming Language Design and Implementation*, 2005, pp. 213–223. [Online]. Available: http://dx.doi.org/10.1145/1064978.1065036
]

[9] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for C,” in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272. [Online]. Available: http://dx.doi.org/10.1145/1095430.1081750

[10] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *USENIX Symp. Operating Systems Design and Implementation*, 2008.

[11] K. Sen and G. Agha, “CUTE and jCUTE: concolic unit testing and explicit path model-checking tools,” in *Int. Conf. Computer Aided Verification*, 2006, pp. 419–423. [Online]. Available: http://dx.doi.org/10.1007/11817963_38

[12] A. Farzan, A. Holzer, N. Razavi, and H. Veith, “Con2colic testing,” in *ESEC/FSE Joint Meeting on Foundations of Software Engineering*, 2013, pp. 37–47. [Online]. Available: http://dx.doi.org/10.1145/2491411.2491453

[13] K. Kähkönen, O. Saarikivi, and K. Heljanko, “LCT: A parallel distributed testing tool for multithreaded Java programs,” *Electronic Notes in Theoretical Computer Science*, pp. 253—259, 2013.

[14] C. Flanagan and P. Godefroid, “Dynamic partial-order reduction for model checking software,” in *ACM Symposium on Principles of Programming Languages*, 2005, pp. 110–121. [Online]. Available: http://dx.doi.org/10.1145/1047659.1040315

[15] T. Boland and P. Black, “Juliet 1.1 C/C++ and Java test suite,” *IEEE Computer*, vol. 45, no. 10, 2012. [Online]. Available: http://dx.doi.org/10.1109/MC.2012.345

[16] J. Jaffar, A. Santosa, and R. Voicu, “An interpolation method for CLP traversal,” in *Int. Conf. Principles and Practice of Constraint Programming (CP)*, 2009, pp. 454–469. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04244-7_37

[17] A. Ibing, “Dynamic symbolic execution with interpolation based path merging,” in *Int. Conf. Advances and Trends in Software Engineering*, 2016.

[18] L. deMoura and N. Bjorner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[19] M. Abadi, C. Flanagan, and S. Freund, “Types for safe locking: Static race detection for Java,” *ACM Trans. Programming Languages and Systems*, vol. 28, no. 2, pp. 207–255, 2006. [Online]. Available: http://dx.doi.org/10.1145/1119479.1119480

[20] J. Voung, R. Jhala, and S. Lerner, “RELAY: static race detection on millions of lines of code,” in *ACM Symp. Foundations of Software Engineering (ESEC-FSE)*, 2007. [Online]. Available: http://dx.doi.org/10.1145/1287624.1287654

[21] P. Pratikakis, J. Foster, and M. Hicks, “LOCKSMITH: Practical static race detection for C,” *ACM Trans. Programming Languages and Systems*, vol. 33, 2011. [Online]. Available: http://dx.doi.org/10.1145/1889997.1890000

[22] M. Eslamimehr and J. Palsberg, “Race directed scheduling of concurrent programs,” in *ACM Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 301–314. [Online]. Available: http://dx.doi.org/10.1145/2692916.2555263

[23] M. Naik, “Effective static race detection for java,” Ph.D. dissertation, Stanford University, 2008.

[24] S. Qadeer and J. Rehof, “Context-bounded model checking of concurrent software,” in *TACAS*, 2005. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31980-1_7

[25] I. Rabinovitz and O. Grumberg, “Bounded model checking of concurrent programs,” in *Int. Conf. Computer Aided Verification (CAV)*, 2005. [Online]. Available: 10.1007/11513988_9

[26] L. Cordeiro and B. Fischer, “Verifying multi-threaded software using SMT-based context-bounded model checking,” in *ACM Int. Conf. Software Eng.*, 2011, pp. 331–340. [Online]. Available: http://dx.doi.org/10.1145/1985793.1985839

[27] P. Godefroid, “Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem,” *Lecture Notes in Computer Science*, vol. 1032, 1996. [Online]. Available: http://dx.doi.org/10.1007/3-540-60761-7

[28] V. Kahlon, C. Wang, and A. Gupta, “Monotonic partial order reduction: An optimal symbolic partial order reduction technique,” in *CAV*, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_31

[29] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Optimal dynamic partial order reduction,” in *ACM Symposium on Principles of Programming Languages*, 2014. [Online]. Available: http://dx.doi.org/10.1145/2535838.2535845

[30] D. Marino, M. Musuvathi, and S. Narayanasamy, “LiteRace: Effective sampling for lightweight data-race detection,” in *ACM Conf. Programming Language Design and Implementation*, 2009. [Online]. Available: http://dx.doi.org/10.1145/1542476.1542491

[31] C. Flanagan and S. Freund, “FastTrack: Efficient and precise dynamic race detection,” in *PLDI*, 2009. [Online]. Available: http://dx.doi.org/10.1145/1542476.1542490

[32] M. Bond, K. Coons, and K. McKinley, “PACER: proportional detection of data races,” in *ACM Conf. Programming Language Design and Implementation*, 2010. [Online]. Available: http://dx.doi.org/10.1145/1806596.1806626

[33] J. Erickson, M. Musuvathi, S. Burchhardt, and K. Olynyik, “Effective data-race detection for the kernel,” in *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[34] A. Itzkovitz, A. Schuster, and O. Mordehai, “Towards integration of data race detection in DSM systems,” *J. Parallel and Distributed Computing*, vol. 59, pp. 180–203, 1999. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1999.1574

[35] E. Pozniansky and A. Schuster, “MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs,” *J. Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.v19:3

[36] Y. Yu, T. Rodeheffer, and W. Chen, “RaceTrack: Efficient detection of data race conditions via adaptive tracking,” in *ACM Operating Systems Review*, 2005. [Online]. Available: http://dx.doi.org/10.1145/1095809.1095832

[37] T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: a race-aware Jave runtime,” *Communications of the ACM*, vol. 53, no. 11, pp. 85–92, 2010. [Online]. Available: http://dx.doi.org/10.1145/1839676.1839698

[38] P. Zhou, R. Teodorescu, and Y. Zhou, “HARD: Hardware-assisted lockset-based race detection,” in *Int. Symp. High-Performance Computer Architecture*, 2007. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2007.346191

[39] J. Devietti, B. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, “RADISH: always-on sound and complete race detection in software and hardware,” in *Int. Symp. Computer Architecture*, 2012, pp. 202–212. [Online]. Available: http://dx.doi.org/10.1145/2366231.2337182

[40] P. Deligiannis, A. Donaldson, and Z. Rakamaric, “Fast and precise symbolic analysis of concurrency bugs in device drivers,” in *Int. Conf. Automated Software Eng.*, 2015. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.30

[41] K. McMillan, “Lazy annotation for program testing and verification,” in *Int. Conf. Computer Aided Verification (CAV)*, 2010, pp. 104–118. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_10

[42] D. Chu and J. Jaffar, “A framework to synergize partial order reduction with state interpolation,” in *Hardware and Software: Verification and Testing*, 2014, pp. 171–187. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-13338-6_14