Triggering Data Races in Multi-Threaded Programs Using Enhanced S2E

Egor Kutovoi
MIPT
Moscow, Russia
e-mail: kutovoi.ea@phystech.edu

Fedor Niskov
MSU, ISP RAS
Moscow, Russia
e-mail: fedor.niskov@ispras.ru

Shamil Kurmangaleev ISP
RAS
Moscow, Russia
e-mail: kursh@ispras.ru

***Abstract*—Most critical software used and developed today is built with the use of multiple threads, which is quite complicated to get right. This leads to a prevalence of bugs related to concurrency, such as data races. In addition to that, current technological trends include robotics and unmanned vehicle software, which is not only multi-threaded but also made more complicated since it interacts with hardware, requires a specific OS environment, utilizes IPC, and must be rigorously tested. In this paper, we build on top of a multi-core version of S2E and provide an algorithm designed to improve the capabilities of detecting data race bugs in programs running in complicated environments. To achieve this, we utilize S2E's full-system emulation, its plugin system, and the CPU core scheduler from our fork.**

***Keywords*—S2E, full-system emulation, symbolic execution, multi-threading, race detection.**

I. Introduction

The complexity of software used and developed is steadily increasing. This can be attributed to the fact that the functional requirements for software are expanding. For example, the demand for highly functional robots and unmanned vehicles is rising. With it, more and more complex software is being developed, which in turn uses new, more efficient hardware, but at the same time harder to handle correctly. Hence, more possibilities for bugs to occur. In particular, this includes multi-core processors and in tandem multi-threaded programs. A substantial number of bugs in multi-threaded programs can be categorized as data race bugs. A data race is said to occur when at least two threads try to access the same shared variable without synchronization and at least one of those accesses is a modification. Due to the inherent complexity that comes with multi-threading, detecting and debugging data races is not an easy task. Because of this, many tools have been developed to help developers find data race bugs. The most notable among them is the famous TSan [1]. However, it is not a perfect solution and does have its limitations. In this paper, we present an algorithm that, given a program built with TSan, collects an original non-failing trace, analyzes it, and constructs an instruction schedule that triggers TSan. We implement our algorithm within our enhanced S2E [2]. It is capable of executing programs within a full-system multi-core emulated environment. By building upon this technology, we can find data race bugs in complex real-world software, including robotics and unmanned vehicles firmware [3].

II. Limitations of TSan

TSan is a dynamic data race detector. It works by instrumenting the target program at compile time, inserting runtime checks near all memory access operations. Memory accesses are evaluated with an efficient algorithm that checks whether the current access races with some other memory access operation. It does this through an algorithm based on the idea of a Happens-Before relation [4]. Let us examine a straightforward example:

```cpp
int x = 0;
int y = 0;
std::mutex mutex;

void Thread1() {
    y += 1;
    {
        std::lock_guard lock(mutex);
        x += 1;
    }
}

void Thread2() {
    {
        std::lock_guard lock(mutex);
        x += 1;
    }
    y += 1;
}
```

Listing 1: Program containing race, not always detectible by TSan

Here, there are 2 shared variables: x and y. Accesses to x are protected by a mutex, but accesses to y are not. Hence, this program contains a data race bug. However, it manifests only when Thread2 is scheduled before Thread1. If the ordering of execution is the other way around, then, as per the definition of Happens-Before, accesses to y are not in a race due to the intra-thread synchronization provided by the mutex and program order. As such, TSan [1] would not trigger.

III. Related Work

Before proceeding to explain our algorithm, let us review the existing approaches for detecting data races in programs. As with any program behavior analysis problem, for data race detection, there exist 3 main approaches: static analysis, when the code is checked without execution; dynamic analysis, when all checks are done during the program's execution; and hybrid approaches that combine the two. Dynamic analysis is definitely the most widely used approach. Tools utilizing dynamic analysis can be further categorized into 3 types: Happens-Before based (DJIT+ [5]), Lock-Set based (Eraser [6], Goldilocks [7]), or a hybrid of the two (TSan [1], Helgrind+ [8]). In our work, we focus on integrating with TSan, in particular, as it is very widely applicable, has been used in production environments for a considerable amount of time, and is quite robust in its detection capabilities. Despite that, TSan, being a dynamic analysis tool, only considers one trace of execution and won't report any bugs that do not reproduce in that particular trace. Our algorithm utilizes both dynamic and static analysis, so it fits into the hybrid category. Considerable research has already been performed in this direction. Most notably, tools like CLAP [9] for simplifying reproduction of data race bugs, Portend [10] for categorizing and simplifying debugging, and Cortex [11] for finding data race bugs given input of production traces. In all of these works, Happens-Before constraints are encoded as predicates and solved via an SMT solver, similar to our approach. However, no tools exist that are capable of finding new data race bugs in a full-system emulated environment, like in our work. In our previous work [2], we have used S2E [12] and enhanced it with support for emulating multi-core OS environments to improve its capabilities for detecting races in multi-threaded programs. In this paper, we build upon these changes and present an algorithm that encourages data races to occur. It utilizes the fact that we can precisely control the execution flow of different emulated CPU cores.

IV. Race Finding Algorithm

A. Idea

In our algorithm, we consider memory read and write instructions to shared variables and mutex lock/unlock operations. Using S2E's plugin system, we intercept the instructions and record the trace. After the trace is fully recorded, we try to change the order of the instructions to find a schedule that would trigger a data race detector, while preserving the existing Happens-Before relations between instructions. To compute the desired order of instructions, we construct predicates and solve them using an SMT solver, Z3 [13] in our case. To keep the computation requirements reasonable, we split the full trace into manageable segments and try to construct an isolated predicate for each particular segment. Each segment is processed in sequence, and if for some segment, we get a satisfiable model, we stop checking further segments. At this point, we dump all of the unchanged trace segments up to the last checked segment, and the reordered last segment. Lastly, we execute the target program for the second time, but now we use the computed schedule within the CPU scheduler, which guides the execution. It should be noted that we run the target program in an environment where each thread is bound to one CPU core exclusively, so the terms "thread" and "core" are mostly interchangeable for our purposes. The program is expected to be built with TSan [1] instrumentation, which then detects the race when it eventually gets triggered. The general flow of the algorithm is as follows:

1) Run the target program inside of S2E while recording the original schedule of instructions S
2) Split the full schedule into manageable segments $s_1 \dots s_m$, for each segment s_i:
   a) Build a predicate for computing a schedule that would trigger a data race
   b) Try to solve the predicate and construct the re-ordered segment s'_i
   c) If it is satisfiable, then save the new schedule S' consisting of segments $s_1 \dots s_{i-1}, s'_i$
3) Run the target program inside S2E for the second time, but guide the execution according to the computed schedule S'.

Let's examine the structure of a predicate constructed in 2.a. Its purpose is to find a permutation of the instructions in the original trace that would trigger a data race, if it is theoretically reachable. As inputs, it takes a set of variables $p(i_j)$ that define a permutation of a set of instructions $\{i_1 \dots i_n\}$. Then, we encode the satisfaction of all Happens-Before relation properties and encode the existence of a data race. In particular, the predicate is a conjunction of 4 parts:

1) $P_{permutation}$: Requirements for $p(i_j)$ to be a valid permutation
2) $P_{core-order}$: Enforcing that the order of instructions withing each core is the same as in the original schedule
3) P_{mutex}: Enforcing that the synchronization provided by mutexes is accounted for
4) P_{races}: Postulating that the segment contains a data race

The computed trace S' is similar in spirit to a combination of a **path predicate** and a **security predicate**:

- Path predicate consists of the unchanged traces $s_1 \dots s_{i-1}$, parts of the predicate used for constructing s'_i: $P_{permutation}$, $P_{core-order}$, P_{mutex}
- And the security predicate is the last part of the predicate for s'_i: P_{races}

Here's how the algorithm would perform on the example given in the beginning:

Sample Run of the Algorithm

Step	Original	Computed	Comment
1	C1 : Read y	C1 : Read y	Uncontended read stays first.
2	C1 : Write y	C2 : Lock M	Core 2's critical section is moved up.
3	C1 : Lock M	C2 : Read x	
4	C1 : Read x	C2 : Write x	
5	C1 : Write x	C2 : Unlock M	
6	C1 : Unlock M	C2 : Read y	**Data Race.**
7	C2 : Lock M	C1 : Write y	
8	C2 : Read x	C1 : Lock M	Original order resumes for Core 1's critical section.
9	C2 : Write x	C1 : Read x	
10	C2 : Unlock M	C1 : Write x	
11	C2 : Read y	C1 : Unlock M	
12	C2 : Write y	C2 : Write y	Final write is now isolated.

Now, let us move on to the next section and formally define how predicates are constructed.

B. Definitions

Firstly, let us define a Happens-Before relation, introduced in [4]. For our purposes, we will be using a definition in terms of instructions executed on the CPU cores of a processor. Each instruction can be one of 4 types: read, write, mutex lock or mutex unlock. Let I be a set of instructions. Let C be a set of CPU cores that execute instructions. Every $i \in$ I is executed by some $c \in$ C. We denote this by $C(i) = c$. Let T $= \{Read, Write, Lock, Unlock\}$ be a set of allowed instruction types. Every $i \in$ I has a type $t \in$ T. We denote this by $T(i) = t$. Let M be a set of messages passed between instructions. Each message is an action of synchronization between CPU cores. For our purposes, we have only one case of synchronization: a mutex lock being taken after a preceding mutex unlock. Happens-Before is a strict partial order relation on the set of instructions, where each instruction is of some type and is executed by some CPU core. The definition is as follows: Let $i_1, i_2, i_3 \in$ I, $c_1, c_2 \in$ C, $m \in$ M

$$\text{Happens-Before}(i_1, i_2) = \text{HB}(i_1, i_2)$$

The following properties hold:

1) If $C(i_1) = C(i_2)$ and i_1 came before i_2, then $\text{HB}(i_1, i_2)$
2) If the event i_1 is the action of sending a message M and i_2 is the action of receiving the message, then $\text{HB}(i_1, i_2)$
3) If $\text{HB}(i_1, i_2)$ and $\text{HB}(i_2, i_3)$, then $\text{HB}(i_1, i_3)$
4) $\neg\text{HB}(i_1, i_1)$
5) If $\text{HB}(i_1, i_2)$, then $\neg\text{HB}(i_2, i_1)$

With this in mind, we can then define what it means when 2 instructions race with each other:

$$\begin{aligned}\text{Races}(i_1, i_2) = \text{R}(i_2, i_2) \iff & T(i_1), T(i_2) \in \{Read, Write\} \\ & \wedge Write \in \{T(i_1), T(i_2)\} \\ & \wedge \neg\text{HB}(i_1, i_2) \wedge \neg\text{HB}(i_2, i_1)\end{aligned}$$

To define the algorithm, we will need a few more definitions. Let us cover them now. Let I $= \{i_1 \dots i_j \dots i_n\}$ be a list of instructions in the order in which they appeared in the original instruction schedule. Let C $= \{c_1 \dots c_N\}$ be the set of all CPU cores. Let A be the set of memory addresses that are associated with instructions. For an instruction $i \in$ I and an address $a \in$ A, we denote $A(i) = a$ if instruction i is associated with the address a. If $T(i) \in \{Read, Write\}$, $A(i)$ has the meaning of an address being accessed by the instruction. If $T(i) \in \{Lock, Unlock\}$, then $A(i)$ has the meaning of the address of the mutex being accessed by the instruction. For instructions $i_j, i_k \in$ I let $p(i_j) = k$ mean that the target permutation p will contain the instruction i_j in the k-th place.

C. Prerequisite

For each given mutex encountered in the schedule, the number of lock and unlock operations on it must be equal.

D. Predicate definition

The goal is to compute the final predicate P. The final predicate is composed of a set of intermediate predicates. For each intermediate predicate, we will give a mathematical definition for constructing it. These definitions then easily translate to C++ code utilizing the Z3 C++ library [14].

$$P = P_{permutation} \wedge P_{core-order} \wedge P_{mutex} \wedge P_{races}$$

Let us now formulate how each intermediate predicate is defined.

1) $P_{permutation}$: Encodes the requirements for a permutation. A permutation is a function that maps a list to a list where each value is unchanged, but may be reordered. As such, it can be encoded as an array of integers, where a value $p(i_j)$ encodes an index where the value i_j will be placed in the resulting reordered list. And thus to enforce that $p(i_j)$ is a valid permutation, we must require that every value is distinct and bounded by the size of the array.

$$\begin{aligned}P_{permutation} = & (\forall j, k \in 1 \dots n, i \neq k : p(i_j) \neq p(i_k)) \\ & \wedge (\forall j \in 1 \dots n : 1 \leq p(i_j) \leq n)\end{aligned}$$

2) $P_{core-order}$: Prohibits mixing the order of instructions withing each core, thereby ensuring property 1 of the Happens-Before relation. To encode this, we consider all instructions for a specific core in the order in which they appeared initially and require that they stay ordered in the same way. And we do this for each core.

$$\begin{aligned}P_{core-order} = & \forall l \in 1 \dots N, \\ & \forall j, k \in 1 \dots n, \\ & (j < k), \\ & (\neg\exists h \in 1 \dots n : j < h < k, C(i_h) = c_l), \\ & (C(i_j) = C(i_k) = c_l) : \\ & p(i_j) < p(i_k)\end{aligned}$$

3) P_{mutex}: Ensures mutex locks and locks constitute a synchronization, and, thus, ensures property 2 of the Happens-Before relation. We do this for each existing mutex independently. Firstly, we count how many lock-unlock pairs there are for a particular mutex. If only one pair exists, then this mutex has not done any meaningful synchronization for the trace segment considered now, and we can skip it. If there is more than one pair, due to how mutexes work, all operations on it would be ordered like so: $Lock \rightarrow Unlock \rightarrow \dots \rightarrow Lock \rightarrow Unlock$. If the total number of $Unlock$ operations is M, then there are $M - 1$ $Unlock$ operations that have a corresponding $Lock$ operation following them. Since we can't deduce which exact $Unlock$ operations have a $Lock$, we encode a predicate for each, but require only $M - 1$ of them to be satisfied.

$$\begin{aligned}
\text{Mutex-Op-Count}_a &= \sum_{j=1}^{n} (A(i_j) = a) \wedge (T(i_j) = Unlock) \\
P_{\text{unlock-has-lock},j} &= (T(i_j) = Unlock)\,, \\
&\exists k \in 1 \dots n, \\
&(T(i_k) = Lock \wedge A(i_k) = A(i_j)) : \\
&p(i_j) < p(i_k) \\
\text{Sync-Ops}_a &= \#\{ \\
&j | j \in 1 \dots n : \\
&P_{\text{unlock-has-lock},j} \wedge A(i_j) = a \\
&\} \\
P_{mutex,a} &= (\text{Mutex-Op-Count}_a = 1 \vee \\
&\text{Sync-Ops}_a = \text{Mutex-Op-Count}_a - 1) \\
P_{mutex} &= \forall a \in \mathrm{A} \exists i \in \mathrm{I} : T(i) = Lock \wedge P_{mutex,a}
\end{aligned}$$

4) P_{races}: Postulates that the schedule will contain a race as per the definiton. Here we utilize the fact that 2 instructions executing on different cores do not have an HB relation $\iff$ they can be reordered sequentially. This is true because we consider only mutexes as synchronization primitives.

$$\begin{aligned}
P_{races} &= \forall j \in 1 \dots n, \forall k \in 1 \dots n, \\
&T(i_j), T(i_k) \in \{Read, Write\}, \\
&Write \in \{T(i_j), T(i_k)\}, \\
&C(i_j) \neq C(i_k) : \\
&p(i_j) = p(i_k) + 1 \vee p(i_k) = p(i_j) + 1
\end{aligned}$$

V. Testing

We have tested our algorithm on sample ROS2 applications injected with bugs, all running in a fully-emulated environment inside S2E. For our tests, we were using ROS2 kilted on Ubuntu 24.04.

Tests summary

Name	TSan finds bug	Our algorithm finds bug	Description
publishers	No	Yes	2 parallel publishers
subscribers	No	Yes	2 parallel subscribers and 1 publisher
service_callers	No	Yes	1 service and 2 parallel callers

VI. Conclusion

Thus, we have presented our algorithm for triggering data races in S2E. It utilizes a full-system multi-core emulator and a controllable CPU core scheduler to quickly trigger rare data races that otherwise could have stayed undiscovered by traditional data race checkers, such as TSan. Our algorithm integrates into our S2E fork and works alongside some dynamic race checker. And we have successfully tested our solution with sample buggy ROS2 applications. Several improvements can be made to this idea, such as supporting more synchronization primitives and more thorough integration with the symbolic engine. We believe that the usage of symbolic execution in conjunction with full-system emulation is a perspective research direction for finding data races in real-world software. This work outlines the possibilities of it, and we are inspired to continue working in this direction further.

References

[1] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," *Proceedings of the Workshop on binary instrumentation and applications*, pp. 62–71, 2009.

[2] F. V. Niskov, E. A. Kutovoy, and S. F. Kurmangaleev, "Enhanced s2e for analysis of multi-thread software," *Program. Comput. Softw.*, vol. 49, no. Suppl 1, S39–S44, 2023.

[3] "The robot operating system." (2025), [Online]. Available: https://www.ros.org/.

[4] 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.

[5] 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, 2003.

[6] 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, 1997.

[7] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race-aware java runtime," *Commun. ACM*, vol. 53, no. 11, pp. 85–92, 2010.

[8] A. Jannesari, K. B., V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient d ynamic r ace d etector," *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–13, 2009.

[9] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," *SIGPLAN Not.*, vol. 48, no. 6, pp. 141–152, 2013.

[10] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: Telling the difference with portend," *SIGPLAN Not.*, vol. 47, no. 4, pp. 185–198, 2012.

[11] N. Machado, B. Lucia, and L. Rodrigues, "Production-guided concurrency debugging," *SIGPLAN Not.*, vol. 51, no. 8, 2016.

[12] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[13] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, Budapest, Hungary: Springer-Verlag, pp. 337–340, 2008.

[14] "Z3 c++ api." (2025), [Online]. Available: https://z3prover.github.io/api/html/namespacez3.html.