MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Extending Concurrency Safety Checking in Slowbeast

Master's Thesis

SUYASH SHANDILYA

Brno, Fall 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Extending Concurrency Safety Checking in Slowbeast

Master's Thesis

SUYASH SHANDILYA

Advisor: prof. RNDr Jan Strejček, Ph.D.

Department of Computer System Security and Communication

Brno, Fall 2024


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Suyash Shandilya

Advisor: prof. RNDr Jan Strejček, Ph.D.

Acknowledgements

I would like to thank my guide prof. RNDr. Jan Strejček, Ph.D. for providing me the opportunity to work on this problem and for his patience, guidance, and trust in my abilities. I would also like to thank my family for their unwavering support along every step of the way. Lastly, I wish to thank my friends for their constant moral support.

Abstract

Slowbeast is an efficient symbolic executor capable of performing various program verification tasks. In this work, we extend its functionality by enabling it to analyse concurrent program for data races. We use a dynamic partial order reduction to reduce the number of interleavings that need to be explored for concurrent program analysis. We compare our results with two state-of-the-art tools for data race detection on relevant benchmarks of SV-COMP 2024. The results indicate the effectiveness and accuracy of our implementation.

Keywords

Symbolic execution, Slowbeast, Data Race, Formal Verification

Contents

1 Introduction

With the advancement of computer science and engineering, our aspirations for solving increasingly complex problems have grown similarly. The resolution of complex issues requires the development of sophisticated software solutions. For example, an error rate of one in a million in an internet security application can result in numerous failures daily, whereas a single error in the code of a rover can cause severe loss to a major space mission if not completely jeopardize it [1]. Formal verification has emerged as a vital technique in computer science for ensuring that programs possess specific desirable properties. Program verification, in particular, involves assessing whether a software application adheres to certain criteria such as memory safety, never reaching an undesirable state, or other domain specific properties.

A crucial programming bug that we want to avoid is the issue of *data race*. A data race occurs when multiple parts of a program simultaneously access the same memory location, with at least one part performing a write operation. This problem has been dramatically highlighted by the notorious bug in the Therac-25 incident [2]. Moreover, data races pose significant threats to data management systems such as MySQL, which are implemented in various sectors and require assurances of data consistency and determinism [3]. Although data races might not always manifest due to the non-randomized and often predictable nature of scheduling, this latent bug represents a substantial risk. Therefore, it is critical to design programs that are free from data races. The verification process essentially validates that concurrent memory access does not occur at any stage where another thread is writing to the same location.

Symbolic execution [4] is a method for running programs on a range of inputs at once. This is useful for finding errors or verifying program correctness. Unlike testing, which only uses specific input and, therefore, explores only one execution path at a time, symbolic execution uses symbolic values to represent arbitrary data in a variable. In general, it tries to explore all the execution paths of a program to verify its correctness. Since there can be an infinite number of such paths in real software, symbolic execution might not finish.

This thesis seeks to enhance the capabilities of the symbolic executor Slowbeast, enabling it to detect data races within C or LLVM programs. In the course of doing this, we have also enhanced its capacity to perform any verification task for a multi-threaded program.

2 Symbolic Execution

Symbolic execution or (Symbolic Interpretation) is a software verification technique where instead of concrete values, we interpret the inputs to a program as symbols and all program variables are thus computed in terms of these symbols. It allows for an examination of the program's execution across the entire range of possible values. This simple method is very effective in performing an exact conclusive analysis if completed. It interprets program execution as a tree-shaped transition system consisting of *symbolic states*. Each symbolic state identifies the current location of the program counter for each thread in the program, the symbolic values of all program variables, and a first-order logic formula called *path condition* that represents the sufficient and necessary conditions on the symbols for execution to arrive at the current location. For concurrent programs, each symbolic state also needs to know which threads are currently active, which are paused, the status of ownership of each mutex lock, which await joins from other threads, and which are ready to be joined. Along with individual program counters, it also maintains a separate call stack for each thread but a common set of shared memory location to maintain the heap and global variable values.

These state values help in knowing the exact transitions and symbolic inputs required to reach a certain state. If a violation is found, we can look at the witness state and know exactly how it can be recreated in real execution.

It starts with an initial symbolic state of the program as the root, which grows into an *execution tree* with consequent symbolic states as its nodes. Each new node denotes the execution of an instruction. The edges of these trees are annotated with predicates on the (symbolic) state variables called *path conditions* that must be satisfied for the child node to be reached. Each instruction is *interpreted* as an additional predicate that it adds to the path conditions that existed until the last edge. For single-threaded non-branching instructions, the path condition added is typically `TRUE`, i.e. no change in conditions. The tree has more than one child node when multiple subsequent program locations are possible. This can happen for two reasons: (a) the program is multithreaded and the parent instruction forks a new thread, and

(b) there is a *conditional jump* on the statement; like an `if` statement in C. In the former case, the node forks two children, each corresponding to the first program location in the main thread and the forked thread, with no change to the path conditions. For the latter, the path conditions are predicates on the symbolic value which need to be satisfied in order for the program to jump at the subsequent state.This offers a tremendous amount of information in analysis, as we now know what domain of values of certain program variables, subsequently program inputs, can lead to the execution of a given program location. If the path conditions can never be satisfied, we know that certain parts of the program can never be reached, and thus do not need to be analyzed.

Although this technique is simple to understand, it has some obvious shortcomings. For one, efficient SMT solvers are needed to operate. Another more pertinent problem is the *path explosion* problem, which refers to the exponentially increasing number of states as more program branching instructions are executed. In a concurrent program with multiple threads, loops, and if-else blocks, one might end up with a huge number of states, all of which need to be kept in memory to be executed. This often leads to incomplete analysis because of computational constraints like memory, time, etc. Nonetheless, symbolic execution has been proven to be a very effective technique in program analysis with real world applications [5] and in software verification competitions where tools using symbolic execution have often outperformed other tools in various verification tasks [6].

Symbiotic[1] is one such tool developed in the Formela laboratory at the Faculty of Informatics of Masaryk University. It is a framework that uses static analysis and program slicing, along with several verification engines like JetKlee[2], and Slowbeast[3]. The former is a fork of a popular symbolic verifier Klee [7] which is written in C++ and is optimised for performance. The latter was developed by Marek Chalupa in Python for fast prototyping of features and algorithms related to symbolic execution. Along with the Classical symbolic execution, it also has experimental features like abstract interpretation, compact symbolic

1. https://github.com/staticafi/symbiotic
2. https://github.com/staticafi/JetKlee
3. https://gitlab.com/mchalupa/slowbeast

execution [8], etc. Symbiotic mainly uses Slowbeast for backward symbolic execution with loop folding. This work discusses our updates to Slowbeast. We hope to finalize its integration with Symbiotic to enhance its capacities as well.

2.1 Slowbeast

Slowbeast takes a C or an LLVM program as input for verification and returns the verification result along with other diagnositc statistics like the number of paths explored, cpu run time, etc. We use its LLVM Parser as it is more robust and the LLVM instruction set is simpler instruction set. For example, a C instruction like `x = y + 2` involves reading `y` and writing to `x`. But its equivalent translation in LLVM looks something like `x1 = load y; x2 = x1 + 2; x = store x3`.Therefore, any memory read corresponds to a `load` instruction, while a write corresponds to a `store` instruction. This simplifies data race detection. Additionally, being LLVM compatible makes its application more versatile as any language which can be translated to LLVM can be verified for data race. Although we test our implementation with C programs, Slowbeast provides the option to use the `llvmlite` module in Python to translate the C program into LLVM with minimal optimisation, before using the LLVM parser.

The support for concurrent programs in Slowbeast was erroneous and limited. While it could parse the major concurrency commands of the default multithreading library in C `pthread`, the updation of the interpreter, instruction handlers, and the representation of concurrency in the symbolic state itself was largely missing. A small contribution of this work has been to add the missing the support and make the representation more consistent with multi-threaded programs.

While the pthread library in C offers many thread synchronization primitives, in this work we only cover the major functions: forking and joining threads (`pthread_create`, `pthread_join`, and locking and unlocking of mutexes: `pthread_mutex_lock` and `pthread_mutex_unlock`. If a program uses other primitives, such as signal variable (`pthread_cond_signal`, etc.), we simply report it as an unsupported program and terminate the verification with the result as unknown.

3 Path explosion in multi-threaded programs

The previously discussed path explosion issue in symbolic execution becomes more severe with multi-threaded programs. States with multiple active threads generally require forking a new state for every active thread. Each of these new states will continue to generate additional states while multiple threads remain active. Additionally, not all of these paths need to be explored, which means that most of the added cost of analyzing the new paths is wasteful. We demonstrate this redundancy in the following pseudocode example.

Program 1 A simple multi-threaded program with data race

```
 1: global int v
 2: procedure MAIN
 3:     t1 = fork(Thread1)
 4:     t2 = fork(Thread2)
 5:     join(t1)
 6:     join(t2)
 7:     exit()
 8: end procedure
 9: procedure THREAD1
10:     v ← 1
11:     exit()
12: end procedure
13: procedure THREAD2
14:     v ← 2
15:     exit()
16: end procedure
```

Consider a possible execution sequence of Program 1, where each step is written as an ordered pair $\langle n, I \rangle$ where n is the thread identifier and I is the corresponding instruction.

$$\langle main, fork(t1)\rangle$$
$$\langle main, fork(t2)\rangle$$
$$\langle Thread\_1, write(v, 1)\rangle$$
$$\langle Thread\_2, write(v, 2)\rangle$$
$$\langle Thread\_2, exit()\rangle$$
$$\langle Thread\_1, exit()\rangle$$
$$\langle main, join(t1)\rangle$$
$$\langle main, join(t2)\rangle$$
$$\langle main, exit()\rangle$$

It can be verified that the above run is valid. Observe that if the 2 instructions coloured in red are swapped it would still be a valid run. However, the outcome of the program (in terms of the final value of v) would be different. While this is a demonstrative example, in general such variable values might be used elsewhere, leading to vastly different end results by simply swapping just two instructions in a run. Our analysis should consider these two runs separately and should verify the properties in both these runs.

Now consider swapping the 2 instructions in green. By definition, it would be a different run as compared to the one written above (since the sequence has changed). It would also be a valid run since such a sequence is allowed by the program, by virtue of the fact that the threads run parallel. However, the end result of the program would still be the same. Thus, the program analysis can just use one of these runs and posit the results for both (or more such equivalent) runs.

Lastly, for completion, consider swapping the blue instructions. Although this would also lead to the same state as the original run, such a swap is not allowed by the given program. *Thread2* must join after *Thread1*. Such a run should not be part of our program analysis.

We thus observe some key things in analyzing concurrent programs using their runs.

- Some instructions exist in what we call the *sequential part* of the program. These can never be swapped as it fundamentally violates the *causality* defined in the program. (blue)

- Some instructions do not have their sequence defined in relation to one another. Some of them can be swapped without affecting the final verification result (green), while others cannot (red).

The number of possible runs we can draw from a program grows binomially as the length of the parallel part - specifically 'green' instructions -increases.

3.1 Partial Order Reduction

This concept of exploring paths redundantly can be understood better if we look at programs as an implicit description of a labeled transition system (LTS) [9], these interchangeable instructions would then correspond to *commutable* transitions in an LTS. The exact definition is available in numerous references [10]. Partial order reduction (*POR*) are formal analsysis techniques that help in determining these commutable transitions in a run, thereby reducing the number of traces required to complete the analysis.

A closely associated notion are Mazurkiewicz traces [11]. Interested readers can refer to the cited literature for a formal definition. The word 'trace' could be a little misleading, as it actually represents an equivalence class of runs of a program. A complete analysis must explore at least one run (also referred to as *interleaving*) per Mazurkiewicz trace.

Dynamic Partial Order Reduction (*DPOR*) is a POR technique that determines these equivalence 'on-the-fly' i.e., as the sequence is being constructed [12]. Consequently, it assesses which runs can be skipped during exploration as an equivalent run would have already been visited previously.

3.2 Source-DPOR

Source-DPOR (or *SDPOR*) was another refinement of the original DPOR algorithm, as it uses source sets instead of persistence sets, which are shorter in size and further reduce the number of traces that are traveled. The algorithm guarantees that it explores an equivalent trace for each maximal run. This equivalence is defined with respect to

a *causal relation* that is central to the algorithm. Intuitively, it describes how different executions affect each others order of occurrence. Before we define it formally, we need to formalize *trace* in the context of symbolic execution and SDPOR.

We define *trace* E as a sequence of program instructions that when executed (starting at the initial program state) lead to some program state s_E. We identify each step in the trace as an *event* tuple $e = \langle p, n \rangle$ where p is the thread identifier, and n is the n^{th} occurrence of the thread p in the execution E. The specific instruction that corresponds to each event needs to be inferred from the program and its symbolic execution. We define $enabled(s_E)$ as the set of threads whose events e can be appended to E such that $E.e$ is also a trace. These would be threads in the state that are *active* (i.e., not waiting on a lock, or a join, etc.) and have some instruction ready to be executed in their program counter. We use $dom(E)$ to denote the set of events within a sequence E. We use $<_E$ to denote the total order between the events in E, i.e., $e <_E e'$ denotes that e occurs before e' in E. We use $E' \leq E$ to denote that the sequence E' is a prefix of the sequence E. A trace can be extended by the next possible event of any of the enabled threads. We will use . to extend or concatenate traces with events and also abuse the notation at times by writing an execution $E.e$ as $E.p$ where p is the thread corresponding to the event e. We can now formally define the causal relation for a given execution sequence as an arbitrary asymmetric, irreflexive, and transitive relation which satisfies the following properties:

1. $\rightarrow_E$ is a partial order on $dom(E)$, which is included in $<_E$.
2. The execution steps of each thread are totally ordered, i.e., $e <_E e' \implies e \rightarrow_E e'$ if e and e' correspond to the events from the same thread.
3. It induces a set of equivalent execution sequences, all with the same causal relation. We use $E \simeq E'$ to denote that E and E' are linearizations of the same causal relation.
4. Let s_E denote the LTS state of the program after executing the trace E. If $E \simeq E'$, then $s_E = s_{E'}$.
5. If $E.w$ is a trace, then $E \simeq E'$ iff $E.w \simeq E'.w$.

6. We use $e \nrightarrow_E e'$ to denote the absence of a causal relation between e and e'. For a thread p, let $next_E(p)$ denote the last event of the trace $E.p$ which should belong to the thread p. If $next_E(p) \rightarrow_{E.p.r} next_{E.p}(r)$ and $next_E(p) \nrightarrow_{E.p.q} next_{E.p}(q)$, then $next_E(p) \rightarrow_{E.p.q.r} next_{E.p.q}(r)$. What this means is that if on a trace we have an event of p causes an event of r, this relation will not be affected if an event q is inserted after p, which is not caused by p.

This causal relation helps us to formalize the concept of two events being in a *race*. Informally, we call two events to be in a race if their order of occurrence in an execution can and should be reversed to explore all traces. For instance, if two threads are trying to acquire the same lock, the program could potentially be in two different states depending on which thread is successful in acquiring the lock first. In order to explore all traces, we therefore must explore the other scenario if such a trace is valid.

Note that the previous *data race* does not have an obvious relation with this race. However, we will consider such events to be in *memory race* because changing the writing order on a shared memory can affect the program (LTS) state. We will specify the exact causal relation in terms of program execution and how this memory race implies the existence of a data race in the next chapter. Formally, we define two events $e, e' \in dom(E)$ to be in a race (denoted as $e \diamond\!\!\!\!\!\!\diamond_E e'$) if they have an immediate causal relation ($e \rightarrow_E e'$ and no event e'' exists in E such that $e \rightarrow_E e'' \rightarrow_E e'$), and their order of execution can be reversed in some trace.

Lastly, we define an anti-reflexive and symmetric *independence relation* $\diamond_E$ as $p \diamond_E q$ between two *threads* p and q in a trace E if $E.p.q$ and $E.q.p$ lead to the same program state. It is equivalent to establishing the absence of causality between the next events of p and q after E.

As mentioned before, the aim of our exploration algorithm is two-fold - to cover all Mazurkiewicz traces, while avoiding equivalent traces. The SDPOR algorithm utilizes a couple of sets, specifically *Backtrack* and *Sleep*, to perform these tasks. The *Backtrack* set suggests ways to explore more traces, while the *Sleep* set does the opposite role of restricting exploration of traces that need not be explored if an equivalent one has been visited before. Analogously, the race relation

updates the *Backtrack* set while the independence relation updates the *Sleep* state.

Before stating the formal algorithm, we define two more entities which will be used in it. The subsequence $notdep(e, E)$ of E, contains all events that occur after e but do not have a causal relation with e. Given an execution trace $E.w$, the set $I_E(w)$ refers to the set of threads which have events in w but no causal predecessor in w. We refer to the prefix of e in E in the algorithm as $pre(E, e)$ $(e \notin pre(E, e))$.

Algorithm 2 Source-DPOR (SDPOR) algorithm

Initially Explore($\langle\rangle, \emptyset$)

```
1:  procedure Explore(E, Sleep)
2:      if ∃p ∈ enabled(s_E) \ Sleep then
3:          backtrack(E) ← {p}
4:          while ∃p ∈ backtrack(E) \ Sleep do
5:              for each e ∈ dom(E) such that e ⊗_{E.p} next_E(p) do
6:                  E' ← pre(E, e)
7:                  v ← notdep(e, E).p
8:                  if I_{E'.e}(v) ∩ backtrack(E') = ∅ then
9:                      add some q' ∈ I_{E'.e}(v) to backtrack(E')
10:                 end if
11:             end for
12:             Sleep' ← {q ∈ Sleep | p ◇_E q}
13:             Explore(E.p, Sleep')
14:             add p to Sleep
15:         end while
16:     end if
17: end procedure
```

The SDPOR is given as a recursive exploration algorithm with the main function Explore(E, *Sleep*). The call within it is made as Explore(E.p , *Sleep*′) where p is a thread in $backtrack(E)$ but not in *Sleep*. The new sleep set *Sleep*′ is updated by truncating *Sleep* to threads that are independent of p (lines 12 to 14) . The function returns if it reaches a terminal state where no more threads are enabled, or if the sleep state has been updated so that all enabled threads are blocked.

The more nuanced part of the algorithm is how it picks threads to add to update the backtrack sets for all execution prefixes that will be revisited eventually (lines 5 to 11). As mentioned above, the idea is to ensure that an event e which is in race with the subsequent event e', occurs before p in some subsequent exploration. The following illustration is an example explaining how the algorithm finds the threads to add to the backtrack set.

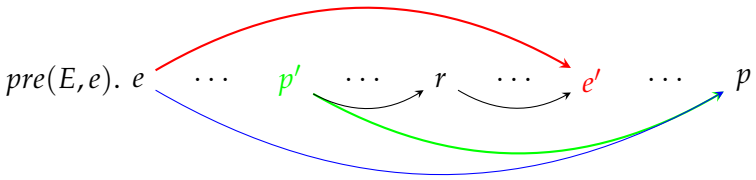

Figure 3.1: Execution trace $E.p$ where e has been found to be in race with p. The arrows denote the causal relation $\rightarrow_{E.p}$ in $E.p$ between the events.

For simplicity, we denote the events by their respective threads. Consider exploring the sequence E and selecting the thread p which will extend it for the next iteration (i.e. E is the entire sequence before p). If we find that some event e in E is in race with p, i.e., $e \neq p$ and e is an immediate causal predecessor of p (shown in the figure with the blue arrow), we need to make sure that our algorithm also explores the trace where p is executed before e, if such a trace is possible.

By the definition of race, we can see that there cannot be any causality between e and p', or between e' and p as the definition of a race disproves the existence of any such event. Clearly, the ordering between p' and p, and between e and e' should also be preserved in the reverse trace. The event r may not cause an event like p' (since then it would be a member of $\{p'\}$) but it can cause e' or p. Thus, along with p', we also need to ensure that r precedes e' and p.

Since we only track the set of threads (and not the order), $backtrack($ $pre(E, e))$ should contain at least one thread from $\{p'\} \cup \{r\}$ that does not have a causal predecessor within it. This idea has already been defined by the set $I_E(w)$. Specifically, we need to choose a thread that has an event in $I_{pre(E,e).e}(notdep(e, E).p)$ and add it to $backtrack(pre(E, e))$. We include p in the argument as the set $notdep(e, E)$ (which is $=$

$\{p'\} \cup \{r\}$ in the illustration) might be empty. If such a thread cannot be found (either $notdep(e, E)$ is empty or $pre(E, e).p$ blocks the execution of e), then the race cannot be reversed and we do not need to worry about missing an informative trace, thus the backtrack is not updated.

In the next chapter, we will explain how we draw the the specific causal relation for this work using symbolic execution and finally how data race is detected.

4 Main Algorithm

In the previous chapter, we looked at how the SDPOR algorithm helps us explore the various traces that will help us visit all possible program states. It does not cover how we compute which events or threads are enabled after a sequence of instructions has been executed. This is a perfect place to use our symbolic executor (Slowbeast).

The most important adjustment was with respect to the executor creating multiple states when executing a branch instruction. Unlike multi-threaded programs, these new states are from the same thread. Fortunately, the SDPOR exploration easily overlays with this behaviour by calling the explore function for each new state (represented by its trace). Our final algorithm is specified in Algorithm 3 where we slightly modify the original SDPOR in Algorithm 2. The data race parts are explained in a later section. For now, we focus on the adjustments made to use symbolic execution.

Algorithm 3 Source-DPOR (SDPOR) algorithm

Initially $data\_race \leftarrow \bot$

```
1:  procedure Run(P)
2:      initial_state ← symexe.get_initial_state(P)
3:      Explore(initial_state, ∅)
4:      if data_race == ⊤ then
5:          Print("Data race")
6:      else
7:          Print("No Data race")
8:      end if
9:  end procedure
10: procedure Explore(s, Sleep)
11:     if data_race == ⊤ then
12:         RETURN
13:     end if
14:     E ← s.trace
15:     if ∃p ∈ enabled(s_E) \ Sleep then
16:         backtrack(E) ← {p}
17:         while ∃p ∈ backtrack(E) \ Sleep do
18:             for each e ∈ dom(E) such that e ⊗_{E.p} next_E(p) do
19:                 if e, p in memory race then
20:                     data_race ← ⊤
21:                     RETURN
22:                 end if
23:                 E' ← pre(E, e)
24:                 v ← notdep(e, E).p
25:                 if I_{E'.e}(v) ∩ backtrack(E') = ∅ then
26:                     add some q' ∈ I_{E'.e}(v) to backtrack(E')
27:                 end if
28:             end for
29:             newstates ← symexe.execute_thread(s, p)
30:             Sleep' ← {q ∈ Sleep | p ◇_E q}
31:             for each s ∈ newstates do
32:                 Explore(s, Sleep')
33:             end for
34:             add p to Sleep
35:         end while
36:     end if
37: end procedure
```

The initial function to call is the RUN() procedure that takes the LLVM program as input. The *symexe* is the main symbolic executor instance whose *get_initial_state* function does all the initial processing of the program and generates the initial symbolic state where the main thread is prepared (setting a call stack, loading the first instruction in the program counter) and global variables are initialized. The function *execute_thread*(s, p) extends the trace of state s by executing the instruction loaded into the thread program counter p. Returns the new set of states which are explored further. In the next section, we discuss how the causal relation is drawn as each new event is added.

4.1 Causal relation in program execution

Given the properties of causality stated before, we can infer the exact causal relations as follows. Note that these only consider immediate causality, as events not matching the following cases could still have transitively causal relations. We will also mention which causal relations are *non-blocking* or *reversible* as those will be the only ones where we need to worry about detecting races.

1. **Same thread instructions:** As a direct result of the requirement, all occurrences of the same thread will be ordered linearly. Each event causally precedes the next event in the same thread, i.e., $\langle p, i\rangle \rightarrow_E \langle p, i+1\rangle$. Note that the occurrence cannot always be inferred from the line number of the instruction because a previous line of code may occur after a later one in the presence of loops or conditional branching. Naturally, this causality is not reversible.

   The following cases are considered only for instructions belonging to different threads.

2. **Lock causality:** Two instructions successfully acquiring the same lock will also have a causal relation to each other. This is introduced to explore the traces where another thread acquires the lock first and runs its instructions. This causality *is* reversible.

3. **Fork causality:** The thread creation event immediately causes the first event of the thread it forks. Naturally, this causality is not reversible either.

4. **Unlock - Lock causality:** The unlock event of a thread should causally precede the event *immediately following* the lock event of another thread acquiring the same lock. The relation is deferred to the next instruction, since regardless of other threads, the locking instruction can be executed and will pause or resume the thread depending on the success of acquisition. This is also not a reversible relation.

5. **Join causality:** Similar to create, the return event of a forked thread causally precedes the event immediately following the join event of the calling thread. This is also a non-reversible relation.

6. **Memory race causality:** If two events belonging to different threads access the same memory location and at least one of them writes to it, they are causally dependent on each other. This is the only other case of reversible causality in this work.

In general, we only consider programs that abide by posix standards and have no undefined behaviours. Double-locking, double-unlocking, or unlocking a lock that has not been acquired are all examples of undefined behavior and should not be considered a valid input for data race verification. Since only the thread that has acquired a lock can unlock it, the unlock event of a thread is always caused by its lock event.

4.2 Data race detection

Once a practical exploration algorithm for symbolic execution of multi-threaded programs has been defined, detecting data races reduces to verifying whether each state satisfies all of the following conditions:

- More than one thread is active.
- The program counter of at least one of the active threads has a `Store` LLVM instruction.

- Another active thread has a `Load` or `Store` instruction accessing the same location as the previous one.

Although checking the above criteria is simple, it turns out that detecting a memory race during exploration automatically implies the existence of a state that exhibits a data race (line 20 of Algorithm 3). This might not be very obvious, since the memory race can exist between two events far apart in a sequence. To understand this implication, let us consider the simple case where consecutive events p and q in a trace $E.p.q$ are in memory race. For simplicity, we refer to both the events and the threads of the events with the same symbols p and q. Clearly, thread q must have been enabled at s_E as well since the memory race instructions cannot block or unblock each other by themselves. Thus, if such a trace can exist, s_E exhibits data race. We will now prove that given a trace $E.p.w.q$ where w is a sequence of events, and and p and q are in a race in the trace, both $E'.p.q$ and $E'.q.p$ exist.

Proof. We have already established that existence of $E'.p.q \implies$ existence of $E'.q.p$, therefore proving one is enough. The proof is trivial when $dom(w) = \emptyset$, so we assume a non-empty w. Since $p \diamond_{E.p.w.q} q$, if there exists an $e \in dom(w)$ such that $p \rightarrow_{E.p.w.q} e$, it implies that $e \not\rightarrow_{E.p.w.q} q$ and vice versa. In other words, any causal successor of p cannot be the causal predecessor of q. If all members of $dom(w)$ are causal successors of p, then q does not depend on any event in w therefore $E.q.p$ is a valid trace. Let p' denote the sequence of events which are not causally preceded by p. If p' is not empty, any event in $dom(p')$ can only be related with q. Thus, the trace $E.p'.q.p = E'.q.p$ is valid. □

Note that the above proof is valid for any kind of race, i.e., both lock and memory race. Thus, we are assured that if concurrent access to a lock or a memory location (with at least one write instruction) is possible, the algorithm will visit a state witnessing such concurrency. But with the memory race and the data race equivalence, we do not have to wait for the exploration to visit the state. This saves a significant amount of time and computation. This is done in lines 19 to 22 in Algorithm 3. Since identifying just one occurrence of a data race is sufficient to consider the program as exhibiting this issue, we activate

the *data_race* flag to true and exit to the procedure RUN(). Conversely, if no race is detected and the exploration is completed successfully, it indicates the absence of data races, and the program is classified as data race free.

5 Implementation and Results

Being a symbolic executor with support for multiple algorithm and features, Slowbeast has multiple modules for each of its methodologies like backward symbolic execution, stateful symbolic execution, etc. Each of these has an interpreter class which handles setting up everything up for execution and exploration, which includes parsing the program, instantiating initial states, along with maintaining the execution tree, running the exploration algorithm, etc. The `IExecutor` class handles the execution of individual instructions and the creation of new states based on the execution. Originally, it created new states for some instructions while updating the existing with the execution for others. This behaviour was made consistent to always create new states instead of updating in place. A wrapper function was introduced for many methods that expected the program counter and call stack to be a state attribute instead of a thread attribute. Many thread synchronization methods incorrectly assumed the origin thread to always be the main thread, which caused some bugs initially and had to be fixed.

The implementation was carried out by forking the existing version of Slowbeast and creating new `IExecutor` and `Interpreter` classes while extending the existing `TSEState` class which supports parsing some `pthread` primitives.

5.1 SV-COMP

SV-COMP is a software verification competition held at the TACAS conferrence [13]. It was launched to fuel the growth of program verifiers and provided a common benchmark set for software verifiers to be evaluated and compared. There are multiple categories of verification such as `Termination`, `Concurrency safety`, etc. with most categories having subcategories that have multiple C programs and a set of LTL properties that the program must verify. To allow for a precise, accurate and reproducible measurement of verifiers, SV-Comp uses a tool called `benchexec` [14]. Each verification task in SV-COMP is run on a machine with a GNU/Linux operating system (x86_64-linux, Ubuntu 22.04) with three resource limits for each run: a memory limit

of 15 GB (14.6 GiB) of RAM, a runtime limit of 15 min of CPU time, and a limit to 4 processing units of a CPU.

Our target verification subcategory was `NoDataRace-Main` under the `ConcurrencySafety` category. At the time of running the experiments, it had a total of 973 valid tasks with 752 expected to have no data race, and the rest 251 having it. For comparing results against other tools, we run our own implementation using `benchexec` on the machines in Formela lab (hostnames `ben01 - ben14`). We did not run the other tools on our own machines, but given the limited resource consumption of our implementation, the comparison is still consistent.

5.2 Goblint and Ultimate Gemcutter

The two verifiers with the highest score in the no data race category of SV-COMP were Goblint [15] and Ultimate Gemcutter [16][1].

5.2.1 Goblint

Goblint uses abstract interpretation for verification and specialises in concurrency safety. It is written in OCaml and is the only verifier in the competition to participate in all categories and did not produce any incorrect result. Goblint also provides a large suite of its regression tests (`goblint-regression`) for benchmarking concurrency safety problems in SV-COMP.

5.2.2 Ultimate Gemcutter

Ultimate Gemcutter (UGemcutter for short) belongs to the Ultimate software analysis framework [17]. It is written in Java and uses the CEGAR method for verification - it starts with an arbitrary trace to choose as a counterexample (*valid* if the trace has data race, *spurious* if the trace is data race free). It proves this trace to be spurious and subsequently generalizes the proof to conclude that a larger (possibly infinite set) of traces are correct. It also uses a similar POR algorithm for reducing the number of evaluated interleavings.

1. Ultimate Gemcutter belongs to the `Ultimate` family of verifiers. Its scores were tied with Ultimate Automiser at 1200. We chose Gemcutter for comparison as it specialises in concurrency safety.

5.3 Results

The final results are tabulated in Table 5.1. Slowbeast here refers to the implmentation in this work. Since the property is defined as `no-data-race` (`LTL(G ! data-race)`), *Correct True* are the number of results for programs that did *not* have a data race and the tool reported it correctly, *Correct False* are the programs that *did* have data race and was detected by the tool. Similarly *Incorrect True* are the results where the programs *did* have data race but the tool reported otherwise, while *Incorrect False* are the benchmarks where the tool reported a non-existent data race.

Table 5.1: Comparison of Slowbeast against state of the art verifiers - Goblint and Ultimate Gemcutter.

	Slowbeast	Goblint	UGemCutter
Correct True	35	669	536
Correct False	26	0	144
Total Correct	**61**	**669**	**680**
Incorrect True	2	0	0
Incorrect False	0	0	1
Total Incorrect	**2**	**0**	**1**
Unknown	931	344	332
Score	32	1338	1200

Each correct True is rewarded 2 points, while correct False gets 1 point. Incorrect False results incur a penalty of -16 points while incorrect True cost -32 points. It can be seen that the final score of our implementation was strongly affected by the two false positive results (64-point penalty). Both of these incorrect results[2],[3] come from a bug that causes incorrect memory location resolution when using arrays.

The biggest reason for the large number of unknown results in our work is the lack of support for *atomic* instrumentation and instructions. SV-COMP uses `__VERIFIER_atomic_begin()` and `__VERIFIER_atomic_end()` in the benchmark programs to model atomic instruction. The instructions between these function calls cannot be inter-

2. `goblint-regression/04-mutex_38-indexing_malloc.i`
3. `04-mutex_37-indirect_rc.i`

rupted. About 430 benchmarks use these function calls. The `weaver` benchmark recently switched to using the `_Atomic` identifiers for variables whose reads and writes will automatically be atomic, i.e., there can be no data race with such instructions being enabled at the same state. There are around 40 such `weaver` benchmarks. Thus, around 470 benchmarks are skipped simply due to a lack of support for modeling atomic instructions. In addition to these, there were around 50 other benchmarks that used pthread APIs that are not supported by our implementation yet, like `pthread_cond_wait`, etc.

In addition to running out of cpu time or memory, the unknown results also comprise a large number of cases where the verifier ran into some error during verification. A leading cause of these runtime errors was our lack of support for the `PTHREAD_MUTEX_INIT` macro, which is used by around 200 benchmarks. It handles the initialization and destruction of mutexes safely, without the need for these explicit instructions. Its implementation is not fully supported in our work. This sometimes causes incorrect resolution of the mutex locks, causing a run-time assertion failure.

5.4 Resource utilization

As noted in the previous chapter, our main algorithm is recursive in nature. Recursion support in Python is limited. The default recursion depth in Python is 3000, but it can be set to any arbitrary amount. We set the recursion depth to 10000, as all the references stated it was a safe enough depth and going higher should be proceeded with caution as it could cause unexpected issues in the Python runtime. About 140 benchmarks stopped because the maximum recursion depth was reached. Although none of our runs consumed more than 5GB of memory and SV-COMP allows up to 15GB, we can experiment with a bigger recursion depth. But more experimentation in a containerized environment is needed as Python does not handle such errors properly.

Figure 5.1 compares the CPU time for correct benchmarks of our implementation against the CPU time of Goblint. For most benchmarks the results are nearly comparable[4] with Goblint being slightly

4. Remember these benchmarks ran on different system thus precise comparison for smaller values is spurious.

Figure 5.1: Scatter plot of CPU Time (in seconds) of Slowbeast (Y-axis) against Goblint (X-axis)

in the lead. For some however, Goblint performs orders of magnitude faster. For instance, the data point in the right corner of Figure 5.1 was computed in 127.82 seconds in slowbeast while Goblint evaluated it in 0.75 seconds.

Against Ultimate Gemcutter however, our implementation is much faster as shown in Figure 5.2. Fig 5.3 is a cropped version of Figure 5.2 added for readability. Except for the data point on the top, all other correct results produced by Ultimate Gemcutter were significantly slower. As an extreme example, the benchmark on the bottom right corner corresponds to a program with data race, which was detected by Ultimate Gemcutter in 557.07 seconds while our implementation finds it in 0.58 seconds. Except for the outlier point at the top, the difference in time between our tool and Ultimate Gemcutter is striking.

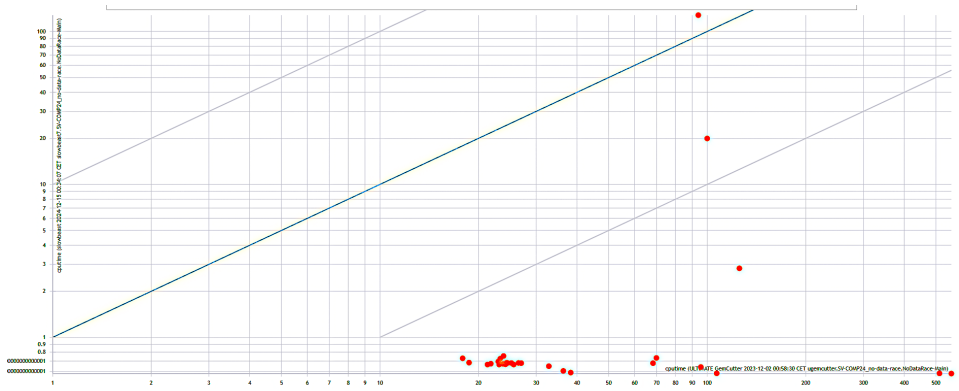

Figure 5.2: Scatter plot of CPU Time (in seconds) of Slowbeast (Y-axis) against Ultimate Gemcutter (X-axis)

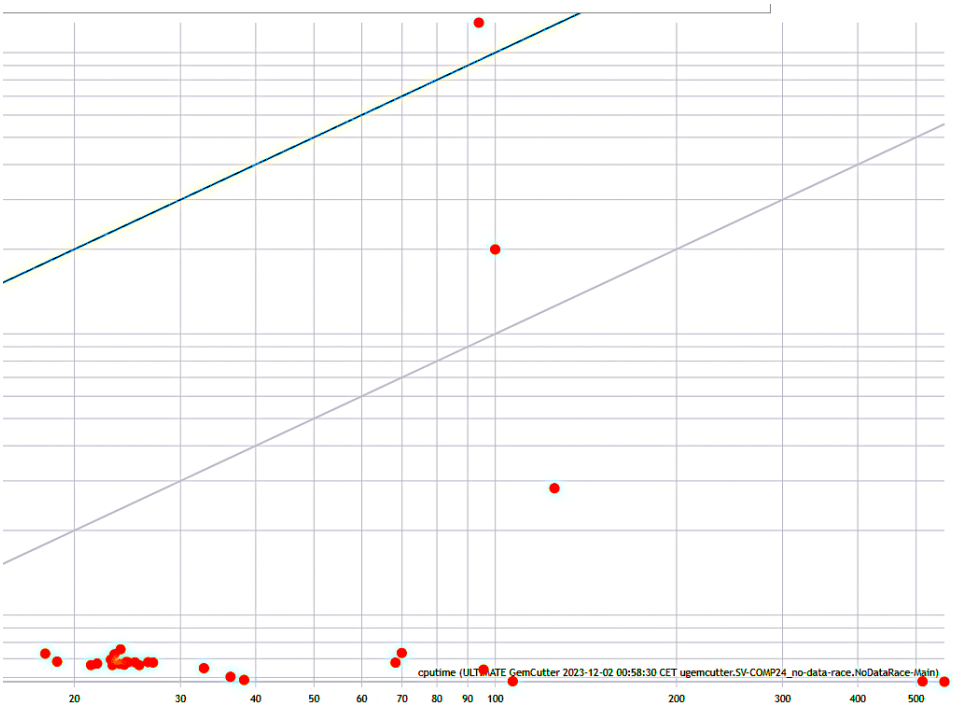

Figure 5.3: Figure 5.2 cropped for clarity

6 Related Work

The main algorithm used here is taken from the paper written by Adbullah et al. in 2014 which introduced an optimal DPOR (ODPOR) along with the SDPOR used here [18]. SDPOR only stores thread identifiers in its backtrack set to explore a trace with race executions reversed. While it never processes two equivalent interleavings, it still suffers from exploring traces which will eventually be sleep set blocked. The optimal DPOR, takes it a step further and stores the sequence of thread identifers as trees, which allows exploration of the reversed race trace, without visiting any sleep set blocked execution. They also demonstrated the efficiency of these algorithms by implementing it on a stateless model checking tool for Erlang called Concuerror. The ODPOR performed slightly better in some tests than SDPOR. However the difference in performance was not large enough to bear the cost of added complexity.

The ODPOR was adapted later to be used along with *Unfoldings* which interpret program execution as a sequence of causally linked *events* [19] (unlike state transitions, as done here). It uses a *conflict relation* to identify noncommuting events for optimal state space exploration. By abstracting transitions as events, they managed to implement a super-optimum algorithm using the conept of *cut-off* events. Simplifying its explanation a bit, cut-off events are identified by comparing the hash of the program memory information contained within the event. Such events always explore the same subsequent states, thus the exploration can be cut short if such an event is visited. They implement their algorithm on a model checking tool written in Haskell called `POET`. It was ran on some modified benchmarks from SV-COMP but the verification tasks and results are not shared.

Later however, D. Schemmel et al. used an enhanced version of this algorithm in KLEE - a well known symbolic executor [20]. The main change from the previous work was that it allowed users to choose any node in the unfolding structure to extend the exploration, instead of the forced DFS ordering. This allowed them to use KLEE's searcher heuristics. Their implementation however, assumed *data-race free* programs. They mention that it can be used to detect data races but their theory does not cover such programs. They ran their tool

on the `unreach-safety` subcategory but not on any `no-data-race` benchmarks.

7 Future Work

Although the results show a positive outlook, it is clear from the benchmarks that the foremost upgrade in the work should be to extend it to support atomic instructions. This will nearly double the number of benchmarks it can run on, allowing us to better assess its effectiveness. To do this, we need to correctly extend the definition of causal relation to include atomic instructions. We also need to upgrade the exploration algorithm so that it is forced to pick the instructions from the same thread, if it is in atomic sequence. It needs to be seen how the backtrack set can be upgraded if a race occurs during an atomic execution.

Another effective upgrade would be to transform the algorithm into an iterative algorithm instead of a recursive one, as it will allow it to better utilize the available memory of the system and not be bridled by the stack capacity of the Python runtime.

Since symbolic execution is an exact analysis technique, we also need to fix the memory resolution bugs in Slowbeast so that the incorrect results can be fixed or at least identified so that no incorrect result is returned.

Another major upgrade would be to correct the interpretation of the `PTHREAD_MUTEX_INIT` macro, as it also blocks many important benchmarks. The implementation is still in its early stage and should be evaluated for performance and optimizations in the future.

8 Conclusion

In this work, we extend the analysis capabilities of Slowbeast to include concurrent programs and detect the existence of data race for supported C and LLVM programs. We use a dynamic partial order reduction technique, specifically Source-DPOR to reduce the number of interleavings that need to be explored for concurrent program analysis. In addition to data race, we can also use this implementation to check for other properties in concurrent programs.

We compare our results on SV-COMP benchmarks and the results indicate the effectiveness and accuracy of our implementation. We compare it with the state of the art verification tools and realize scope of improvement. Although there were a couple of incorrect results in our tests, none of them came from the additions made in this work. We note that the memory location analysis of Slowbeast needs to be improved to avoid returning incorrect results.

We also mention related work that can be used to further the development made in this work.

We note the other ways in which the utility of our work can be improved. We realize that extending our implementation to support atomic instructions should be the most efficient way forward, as it will also allow better testing of our work.

Bibliography

1. JONES, Mike. *What really happened on Mars Rover Pathfinder* [https://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html]. 1997.
2. LEVESON, Nancy G.; TURNER, Clark S. An Investigation of the Therac-25 Accidents. *Computer*. 1993, vol. 26, no. 7, pp. 18–41. Available also from: https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/therac25.pdf.
3. CORPORATION, Oracle. *MySQL Bug #77862: Race condition causes database corruption* [https://bugs.mysql.com/bug.php?id=77862]. 2015.
4. KING, James C. Symbolic execution and program testing. *Commun. ACM*. 1976, vol. 19, no. 7, pp. 385–394. ISSN 0001-0782. Available from DOI: 10.1145/360248.360252.
5. THIEDE, Christoph. *Symbolic Execution and Applications*. 2022. Available also from: https://linqlover.github.io/symbolic-execution-survey/report.pdf.
6. BEYER, Dirk. Progress on Software Verification: SV-COMP 2022. In: FISMAN, Dana; ROSU, Grigore (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2022, pp. 375–402. ISBN 978-3-030-99527-0.
7. CADAR, Cristian; DUNBAR, Daniel; ENGLER, Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. San Diego, CA: USENIX Association, 2008. Available also from: https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems.
8. KUMOR, Kristián. *Compact Symbolic Execution in Slowbeast* [*online*]. 2024. Available also from: https://is.muni.cz/th/d253l/. Master's thesis. Masaryk University, Faculty of Informatics, Brno. SUPERVISOR : Jan Strejček.

9. DERRICK, John; BOITEN, Eerke. Labeled Transition Systems and Their Refinement. In: *Refinement: Semantics, Languages and Applications*. Cham: Springer International Publishing, 2018, pp. 3–26. ISBN 978-3-319-92711-4. Available from DOI: 10.1007/978-3-319-92711-4_1.

10. GODEFROID, Patrice. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

11. MAZURKIEWICZ, Antoni. Trace theory. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency: Advances in Petri Nets 1986, Part II Proceedings of an Advanced Course Bad Honnef, 8.–19. September 1986*. Springer, 1987, pp. 278–324.

12. FLANAGAN, Cormac; GODEFROID, Patrice. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*. 2005, vol. 40, no. 1, pp. 110–121.

13. FINKBEINER, Bernd; KOVACS, Laura (eds.). *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part III*. Vol. 14035. Springer Cham, 2024. Lecture Notes in Computer Science. ISBN 978-3-031-57255-5. Available from DOI: 10.1007/978-3-031-57256-2.

14. BEYER, Dirk; LÖWE, Stefan; WENDLER, Philipp. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*. 2019, vol. 21, no. 1, pp. 1–29. ISSN 1433-2787. Available from DOI: 10.1007/s10009-017-0469-y.

15. SAAN, Simmo; ERHARD, Julian; SCHWARZ, Michael; BOZHILOV, Stanimir; HOLTER, Karoliine; TILSCHER, Sarah; VOJDANI, Vesal; SEIDL, Helmut. Goblint: Abstract Interpretation for Memory Safety and Termination. In: FINKBEINER, Bernd; KOVÁCS, Laura (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer Nature Switzerland, 2024, pp. 381–386. ISBN 978-3-031-57256-2.

16. FARZAN, Azadeh; KLUMPP, Dominik; PODELSKI, Andreas. Sound sequentialization for concurrent program verification. In: JHALA, Ranjit; DILLIG, Isil (eds.). *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 2022, pp. 506–521. Available from DOI: 10.1145/3519939.3523727.

17. HEIZMANN, Matthias; HOENICKE, Jochen; PODELSKI, Andreas. Refinement of Trace Abstraction. In: PALSBERG, Jens; SU, Zhendong (eds.). *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*. Springer, 2009, vol. 5673, pp. 69–85. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-642-03237-0_7.

18. ABDULLA, Parosh; ARONIS, Stavros; JONSSON, Bengt; SAGONAS, Konstantinos. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*. 2014, vol. 49, no. 1, pp. 373–384.

19. RODRIGUEZ, César; SOUSA, Marcelo; SHARMA, Subodh; KROENING, Daniel. Unfolding-based partial order reduction. *arXiv preprint arXiv:1507.00980*. 2015.

20. SCHEMMEL, Daniel; BÜNING, Julian; RODRIGUEZ, César; LAPRELL, David; WEHRLE, Klaus. Symbolic partial-order execution for testing multi-threaded programs. In: *International Conference on Computer Aided Verification*. Springer, 2020, pp. 376–400.

A Attached source code and results

The implementation can be found in the attached zip file `slowbeast-no-data-race.zip`[1]. It also contains the final executable used to run the benchmarks. The `README.md` document outlines the procedure for running the executor independently, specifies the files that were modified, and includes the `benchexec` configuration files. The vericloud could not be configured in time correctly to run the implementation, thus `results.zip` contains the final results files indexed via their respective host pcs (`ben07` corresponds to `testslowbeast7.*` and so on). The final result has been aggregated using the individual runs. The file `res_xtractor.py` in `results.zip` helps further parse the results to identify the different reasons for the unknown results. The `table-generator` tool provided by benchexec can be used to create the scatter plots.

1. https://gitlab.fi.muni.cz/xshandil/slowbeast-no-data-race