RESEARCH-ARTICLE

Data Race Detection through Vibe Translation

JAN HUECKELHEIM, Argonne National Laboratory, Lemont, IL, United States

VIMARSH SATHIA, University of Illinois Urbana-Champaign, Urbana, IL, United States

SIYUAN BRANT QIAN, University of Illinois Urbana-Champaign, Urbana, IL, United States

Open Access Support provided by:

University of Illinois Urbana-Champaign

Argonne National Laboratory

PDF Download
3731599.3767363.pdf
30 December 2025
Total Citations: 0
Total Downloads: 1236

Published: 16 November 2025

Citation in BibTeX format

SC Workshops '25: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis
November 16 - 21, 2025
MO, St Louis, USA

Conference Sponsors:
SIGHPC

Data Race Detection through Vibe Translation

Jan Hueckelheim
Argonne National Laboratory
Lemont, IL, USA
jhueckelheim@anl.gov

Vimarsh Sathia
University of Illinois at Urbana-Champaign
Champaign, USA
vsathia2@illinois.edu

Siyuan Brant Qian
University of Illinois at Urbana-Champaign
Champaign, USA
siyuanq4@illinois.edu

Abstract

We propose a data race detection approach for code written in a source programming language, by means of AI-agent translation to a target language, followed by conventional tool-based detection in the target language. We evaluate this *translate-then-check* approach by translating the C/Fortran+OpenMP programs in DataRaceBench to the Go programming language, and using the Go data race detector to check for races. The translation is controlled through natural language prompts, similar to approaches popularized as *vibe coding*. Translate-then-check achieves 92.8% accuracy and 9 false negatives for the C programs in DataRaceBench, compared to 89.9% accuracy and 17 false negatives for Clang with ThreadSanitizer and Archer applied to the original C programs. We discuss the approach and its overall accuracy, and show examples where translate-then-check leads to false negatives or positives due to limitations of the Go data race checker or, in some cases, limitations of the translation.

CCS Concepts

• **Software and its engineering → Parallel programming languages**; **Formal software verification**; **Software testing and debugging**.

Keywords

Data Race Detection, Go, OpenMP, Vibe Coding

ACM Reference Format:
Jan Hueckelheim, Vimarsh Sathia, and Siyuan Brant Qian. 2025. Data Race Detection through Vibe Translation. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767363

1 Introduction

Data races— defects caused by unsafe concurrent access to the same memory address by multiple threads—are among the most challenging software bugs to identify and fix, due to their inherently non-deterministic nature. Effective data race detection tools can be essential for developing and maintaining parallel software, which is increasingly important in the presence of more diverse hardware platforms, including CPUs, GPUs, and AI accelerators, for which a variety of programming environments and programming frameworks poses programming and portability challenges.

However, the tool landscape for data race detection varies between different programming languages and different parallelism dialects. To bridge this language gap, we propose an approach that leverages AI-driven program translation, with the goal of extending the applicability of mature data race detection tools in one language to programs written in another language. Unlike approaches that rely purely on large language model (LLM)-based race detection, which requires complex reasoning that current LLMs struggle with, we merely use LLMs for language translation, a task at which they perform reasonably well [19]. This is then combined with the use of conventional data race detection tools, hopefully leading to more rigorous and explainable results than a pure AI approach. We refer to this combination of LLM-based translation and conventional race detection as *translate-then-check*.

In the context of dynamic data race detection, where a small ratio of false positives or false negatives can be acceptable, our experiments show that the uncertainty introduced by LLM-based translation can be small enough to result in data race detection rates that are competitive with state of the art tools applied directly to their native language. On the other hand, our experiments also show that LLM-based translation does introduce new race conditions at a rate that may be unacceptable outside of a translate-then-check approach, especially if the translated programs are to be used in production environments. For example, even a powerful state of the art model such as Claude 4 Sonnet sometimes fails on basic tasks such as parallelizing a for loop, while most of the time performing well even for much more complicated tasks. Our results thus also act as a temperature check on the state of LLM-based translation for parallel codes, complementary to other work reporting success using LLMs for translation tasks in applications [10].

We believe the contributions of this paper to be:

- A novel translate-then-check approach, which is to our knowledge the first time that programs are translated by an LLM using natural language prompts, with the sole intent to check for data races in the original source code.
- Evaluation of LLM-based translation from C or Fortran with OpenMP, to the Go language, for the programs contained in DataRaceBench.
- A Go version of DataRaceBench – albeit with inaccuracies introduced by the translation, as described in this paper – available on GitHub[1].
- Improvements to DataRaceBench Fortran programs, due to defects that were found during the translation and while reproducing results.
- A point of comparison for the relative quality of data race detectors across C/OpenMP and Go.

[1]https://github.com/jhueckelheim/dataracebench-vibe-check

Some of the key findings include:

- Translate-then-check for DataRaceBench[13, 14] C/C++ programs translated to Go outperforms the current version of Clang/ThreadSanitizer with Archer [1], itself a mature tool chain.
- For DataRaceBench Fortran programs, both our translate-then-check approach and ThreadSanitizer+Archer perform worse than they do for the C/C++ programs, but our approach still outperforms the alternative.
- The quality of LLM-based translations is crucial for good detection rates. In particular, all false positives in our approach are caused by mistranslations that introduced actual races, and less powerful language models have significantly higher failure rates.

A significant limitation of our work, is the reliance on closed, commercial AI models – specifically Claude 4 Sonnet, accessed through a paid plan with Cursor – which means that our results represent a snapshot in time and may not be reproducible as models evolve. Additionally, DataRaceBench's prominence may have influenced the AI agent's translation decisions, as the benchmark was almost certainly included in the training data, and the nature of each file (race or no race) is also clearly indicated as part of the file name and in comments within the files. While the latter part could have been fixed by removing comments or scrambling file names, the well-known nature of the DataRaceBench programs would still potentially bias studies based on this benchmark, and future work should investigate this approach in a real-world setting where the investigated codes have unknown data race status.

While we acknowledge that users might still have good reasons to prefer native tools when available instead of translate-then-check, we postulate that our experiments nevertheless indicate that translate-then-check can allow the use of mature race detection tools across languages, with an accuracy that is high enough to be useful. This could be particularly interesting for less commonly used or novel source languages or parallelism frameworks, for which mature tools such as Archer, Thread Sanitizer, or the Go race detector may not (yet) exist.

The remainder of the paper is organized as follows: Section 2 discusses relevant work and concepts used in this paper. Section 3 describes our approach in more detail. Section 4 provides an overview of our experimental results, while Section 5 looks at selected programs in more detail and provides a better intuition for the failure modes in our approach, followed by a conclusion in Section 6.

2 Background

We briefly summarize dynamic data race detection, DataRaceBench, and OpenMP and Go language features relevant to this work.

2.1 Dynamic data race detection

A *data race* can occur when two threads access the same memory location concurrently without proper synchronization, and at least one access is a write [20]. Data races can lead to nondeterministic bugs that are notoriously hard to reproduce and debug. Dynamic data race detection tools [2, 8, 11, 15, 21] monitor a particular program execution to catch runtime conditions that cause data races in a specific run, typically by analyzing an execution trace. In contrast to static analysis approaches [3, 7, 16], dynamic detectors analyze a concrete execution, which may miss data races not revealed by that run, but tend to offer higher detection accuracy and fewer false-positives. Most dynamic data race detections are based on lockset-based, happens-before-based, or hybrid [17] techniques. A technique worth mentioning in the context of this work is purely LLM-based detection of data races [6], although this typically works without actually executing the code, and is therefore more closely related to static analysis than dynamic data race detection.

2.2 DataRaceBench

DataRaceBench [13] is a benchmark suite designed for evaluation of data race detection tools. In version 1.4.0 it provides a collection of 208 C programs and 168 Fortran programs that use a variety of OpenMP features. Previously published comparisons of dynamic data race detection tools in [13] are based on a subset of 166 Fortran programs. Roughly half of the programs contain a data race, whose nature and exact code location is contained as comments in the source file. File names for programs with or without race end with `-yes` and `-no`, respectively. We evaluate our approach based on DataRaceBench's most recent C, and most recent as well as older versions of Fortran microbenchmarks.

2.3 OpenMP

We briefly summarize some OpenMP language features used in DataRaceBench programs. For a more comprehensive and precise description we refer to [18]. OpenMP allows programmers to control multi-threaded execution through the use of pragmas, such as `#pragma omp parallel`, which creates a team of threads to execute the following region in parallel. Within a parallel region, the work sharing construct `#pragma omp for` can be used for parallel execution of loop iterations, while `#pragma omp single` specifies that the following code block is executed by only one of the threads. Variables are shared by default (with some exceptions, such as loop counters of work sharing loops), meaning that there is one instance accessible by all threads. Users can also privatize variables, for example using `private` or `firstprivate` clauses, both of which result in each thread having its own instance of this variable (the clauses differ in the way private instances are initialized). OpenMP explicit tasks (created via `#pragma omp task`) specify units of work that can be run by threads asynchronously. OpenMP provides synchronization mechanisms between tasks, including `depend` clauses to specify data dependencies, and `critical` or `taskwait` directives. OpenMP's SIMD directive (`#pragma omp simd`) enables SIMD execution of loop iterations, for example using vector instructions.

In Listing 1, an excerpt from a DataRaceBench program that contains a data race, a team of two threads is created on line 1 with a `parallel` clause. Loop iterations are mapped to the two threads (lines 3–6). Only one thread reaching the `single` clause (line 7) continues while the other thread waits at the implicit barrier, but is allowed to execute other tasks from the task queue. Inside the `single` block, line 9 spawns a parent task, which spawns a child task on line 11. Because `taskwait` on line 16 only waits for the parent task, it does not wait for the child task on line 11. If the child task has not finished by the time line 17 executes, the read of psum[1] races with the child's write on line 12.

```
 1 #pragma omp parallel num_threads(2)
 2 {
 3   #pragma omp for schedule(dynamic, 1)
 4   for (int i=0; i < 4; ++i){
 5     // ... code initializing `a`
 6   }
 7   #pragma omp single
 8   {
 9     #pragma omp task // parent task
10     {
11       #pragma omp task // child task
12       { psum[1] = a[2] + a[3]; }
13
14       psum[0] = a[0] + a[1];
15     }
16     #pragma omp taskwait // wait for parent task only
17     sum = psum[1] + psum[0]; // race on psum[1]
18   }
19 }
```

Listing 1: C/OpenMP program with a data race between lines 12 and 17, due to incorrect task synchronization. The code is abbreviated from DRB117-taskwait-waitonlychild-orig-yes.c

2.4 Go

We summarize selected Go features that are commonly used by the AI agent to implement the translated DataRaceBench programs. Go expresses parallelism in the form of *goroutines*. A function `f (args)` can be launched as a goroutine by writing the statement `go f(args)`. A goroutine can be understood as a lightweight task, which is scheduled for execution by a thread pool that is managed by the Go runtime environment. Due to their lightweight nature, goroutines are in practice often spawned in numbers that far exceed the available processor cores. For example, a parallel loop is often expressed as a sequential loop in which each iteration spawns a new goroutine, see lines 4-9 in Listing 2. Go provides an extensive library of synchronization constructs in its `sync` package, including atomic operations, locks, condition variables, semaphores, and barriers, all of which are common in other parallel languages.

Additionally, Go provides *WaitGroups*. WaitGroups are barrier-like objects, where the number of participating workers can be dynamically adjusted using the `Add(num int)` method, see line 3. A call to the `Done()` method signals arrival at the barrier, while a call to `Wait()` prevents departure until the number of Done calls is equal to the sum of the numbers passed into the preceding Add calls, see lines 6 and 10.

Finally, Go programs commonly use *channels* to implement a paradigm akin to message passing. A channel implements a FIFO queue initialized to a fixed capacity (capacity is 1 if not explicitly specified, see line 12). Sending a message to a channel `c` is expressed as `c<-`; this operation blocks if the channel has no available capacity. Receiving is expressed as `<-c`; this operation blocks until a message is available to be received. Channels are often used to implement synchronization between threads, for example in the form of a boolean channel that is being sent to in line 17, and received from in line 20, where the returned value is simply discarded.

```
 1 var initWg sync.WaitGroup
 2 initWg.Add(4)
 3 for i := 0; i < 4; i++ {
 4   go func(idx int) {
 5     defer initWg.Done()
 6     // ... code initializing `a`
 7   }(i)
 8 }
 9 initWg.Wait()
10
11 parentDone := make(chan bool)
12 go func() { // parent
13   go func() { psum[1] = a[2] + a[3] }() // child
14   psum[0] = a[0] + a[1]
15   parentDone <- true
16 }()
17
18 <-parentDone // wait for parent only
19 sum = psum[1] + psum[0] // race on psum[1]
```

Listing 2: Go program, translated and parallelized by AI from DRB117-taskwait-waitonlychild-orig-yes.c using idiomatic Go features, containing the same error as the original C code

2.5 ThreadSanitizer and the Go race detector

ThreadSanitizer [21] is a widely used open-source (part of the LLVM [12] and GCC [9] sanitizers) dynamic data race detector. ThreadSanitizer maintains a global state to record the observed synchronization events and per-ID states to record information about each memory location of the program. ThreadSanitizer instruments the program to intercept memory access and synchronization events, and feeds these events into a runtime state machine that updates global and per-ID states in real time. By default, Archer is used to detect and instrument OpenMP constructs, which improves the accuracy of ThreadSanitizer in LLVM. We use ThreadSanitizer+Archer as a baseline for comparisons in this work, and refer to this combination simply as TSan.

The Go data race detector is part of the Go distribution, and can be enabled using the `-race` command line argument during compilation or when running a program directly from source code without pre-compilation. The Go data race detector is based on TSan, but well integrated and tailored to the Go language and runtime environment. For example, it can detect races between goroutines even if those goroutines are scheduled for execution on the same thread by the Go runtime environment. The Go data race detector is frequently used in large industrial applications [5].

3 Approach

3.1 Translate-then-check

For the code translation from C or Fortran to Go, we used Cursor[2] and selected the Claude 4 Sonnet model. At the time of writing, this requires a paid plan. We also attempted to use the "Auto" mode, available with a free plan, which internally selects a model in an opaque way that can not be controlled or easily queried by the user. Auto mode however resulted in poor translations that did not capture the semantics of the original programs accurately, leading to twice as many false positives and false negatives in preliminary experiments. We therefore focus only on Claude 4 Sonnet.

We used prompts such as the following to initiate the translation:

> For each file in micro-benchmarks, create a translation to the Go programming language and store the result in the translated/c-to-go folder. Translate as faithfully and closely as possible, without trying to make the programming idiomatic to the Go language. Do not attempt to fix bugs while translating.

In both the C and Fortran cases, the AI agent noticed the large number of files and attempted unsuccessfully to create a script to automate the translation, for example using the `sed` command and regular expressions to replace certain patterns. In order to force translation by the language model itself, we used additional prompts, and eventually a `.cursor` file, which allows additional prompts that always remain in context. Because Claude 4 Sonnet is called as an agent through Cursor, it has access to all files in the repository and is able to use terminal commands for debugging. This was essential, for example, because some of the original Fortran programs declare unused variables, which appeared in the first versions of translated Go files. However, unused variables are treated as an error by the Go compiler, which the language model found when trying to compile and run the generated code. The errors were automatically fixed by the model without user intervention. It is thus not surprising that all translated programs compile and run. This is even true for DRB071 and DRB152, where the original Fortran programs contain, respectively, an uninitialized variable that is used as array sizes, and an uninitialized lock. For DRB071 the AI agent notices this, and arbitrarily chooses a size, and generates the following Go code with comment:

```
length = 100 // Initialize length (was uninitialized
    in Fortran - undefined behavior)
```

This behavior can be seen as a strength and weakness of our approach, as will be discussed in more detail in Section 5, as the translation is often useful, but not always faithful.

The translated files are executed with Go version 1.24.5, with data race detector enabled. We use a script that calls each translated program once for each configuration, using the environment variable `GOMAXPROCS` to control the number of threads, using the same set of thread counts used by the benchmark script in the original DataRaceBench. We also use the `GORACE="halt_on_error=1"` environment variable to terminate execution as soon as a race is found. We combine this with the `timeout` command from `coreutils` to limit the time spent on each program to 5 minutes. None of the programs exceeded this time in our experiments.

[2]https://cursor.com/home

3.2 Why not just ask an LLM?

Previous work [6] has discussed directly asking a language model whether a given code contains a data race. Claude 4 Sonnet achieves 100% accuracy for this task on DataRaceBench. However, as one might suspect, this is because the AI agent uses the file names and comments as hints, as it openly admits in its answer.

> I can determine this from the file names (which contain "yes" or "no" to indicate race presence) and from the comments in the files that specify the data race pairs and line numbers.

For unknown codes, the ability of language models to reason about their behavior and statically detect data races is still an open and evolving question.

4 Experimental Evaluation

To guide our evaluation, we ask the following research questions:

- **RQ1** *Effectiveness of Translation-Based Detection.* How effective is data race detection when applied directly to code translated from C/Fortran to Go?
- **RQ2** *Effectiveness of Combined Analysis.* To what extent does a combined analysis of both native and translated code improve detection performance?

4.1 Experimental Setup

Our experimental setup uses programs from the DataRaceBench (DRB)[13, 14] suite version 1.4.0. We organized our evaluation around 3 distinct configurations.

C/C++. : This configuration contains all 208 C/C++ programs from DRB 1.4.0, serving as a baseline. We use LLVM 21 to reproduce experimental results. We construct instrumented programs by compiling the source code with `clang` using (`-fsanitize=thread`), along with along with `-O3 -march=native` optimization flags and OpenMP support (using `libomp`).

F-old. : The **F-old**(or **Fortran-old**) configuration includes 142 Fortran programs from the original DRB paper's Fortran microbenchmark set that were successfully compiled with `gfortran` 13.3.0 linked against `libomp` and `clang`'s ThreadSanitizer runtime. This setup deliberately excludes about 15% of the original DRB benchmarks that use the `!$omp target` directive, as that feature is implemented by `gfortran` using `libgomp` calls, which is incompatible with `clang-21`'s `libomp` runtime.

F-new. : The **F-new** (or **Fortran-new**) configuration was created to use an end-to-end pipeline based on `flang`[4], a new LLVM-based production Fortran compiler. Since `flang` does not support ThreadSanitizer flags natively, we first emit LLVM-IR using the `-emit-llvm` flag. We then annotate all LLVM functions in the generated IR with the `sanitize_thread` attribute, followed by invoking ThreadSanitizer directly on the generated IR using `llvm-opt`. This configuration includes 133 supported Fortran programs.

We also implemented Fortran fixes for uninitialized variables in microbenchmarks DRB071 and DRB152 and use fixed microbenchmarks for evaluation.

As in DRB [13], we parametrize each benchmark by the number of OpenMP threads from a list of (3, 36, 45, 72, 90, 180, 256), and for var-length set, by array sizes from (32, 64, 128, 256, 512, 1024). For each test configuration, we run the instrumented binary three times. We report our experimental results in Table 2.

4.2 Evaluation Metrics

To evaluate the performance of each data race detection approach, we reuse the 4 standard metrics from DRB: precision, recall, accuracy and F1 score. The mathematical formulas for these metrics are defined in Table 1. Each metric provides a different insight into a tool's performance:

- **Precision** measures the fraction of reported data races that are genuine. A tool with high precision is trustworthy, as it minimizes the number of false positives (FP), saving time from investigating phantom bugs.
- **Recall** measures the fraction of all existing data races that the tool successfully identifies. High recall is critical for effectiveness, since it indicates a low rate of false negatives (FN) and ensures real bugs are not overlooked.
- **F1 Score** is a harmonic mean of Precision and Recall. It provides a balanced measure of the tool's performance, as there is a tradeoff between catching every bug (high recall) vs being sure every bug is real (high precision)
- **Accuracy** measures the fraction of correct predictions across all testcases.

4.3 RQ1: Translation Effectiveness

To answer this question, we evaluate the effectiveness of translating C/C++ and Fortran code to Go and then using Go's native race detector. The evaluation results are presented in Table 2.

For **C/C++ programs**, the translation to Go proves to be highly effective. As seen in Figure 1, the analysis of translated Go code achieves a higher F1 Score(0.926) compared to the baseline analysis using `clang` and TSan(0.892). This improvement is driven by an increase in recall, indicating that Go's data race detector is able to detect more true data races.

For **Fortran programs**, the translated Go code results in lower F1 scores than two baseline Fortran toolchains (**F-old** and **F-new**) due to an increased number of false positives, as seen in Figure 2. However, our approach uncovers more true data races than both baseline Fortran toolchains, achieving substantially higher recall.

4.4 RQ2: Effectiveness of Combined Analysis

We investigate whether a combined analysis, aggregating reports from the native tool and the Go race detector on translated code, can yield better results. Our findings, summarized in Table 2, indicate that a strategy that flags programs in which either tool reports a race, is very effective. This is in part caused by the fact that there is little overlap between the programs misclassified by the tools.

As seen in Figure 1, for **C/C++ programs**, the combined approach (Go $\wedge$ C/C++) achieves a **F1 score** of 0.935, the highest of any method. This is mainly achieved by maximizing recall while

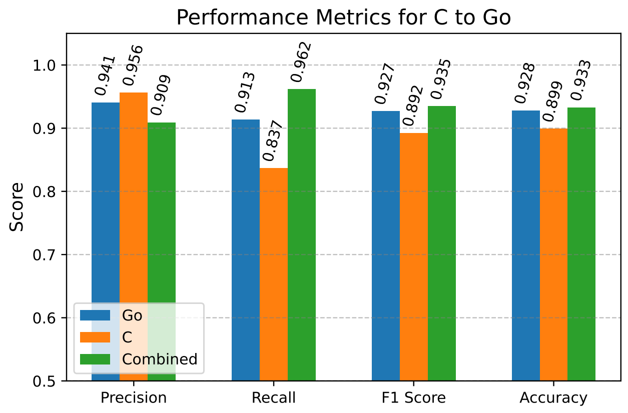

Figure 1: Data race detection performance for C/C++ with OpenMP code translated to Go. We compare `clang`'s ThreadSanitizer based checker, Go's builtin Data Race Detector and a combined approach. The combined approach achieves the highest F1 Score of 0.935.

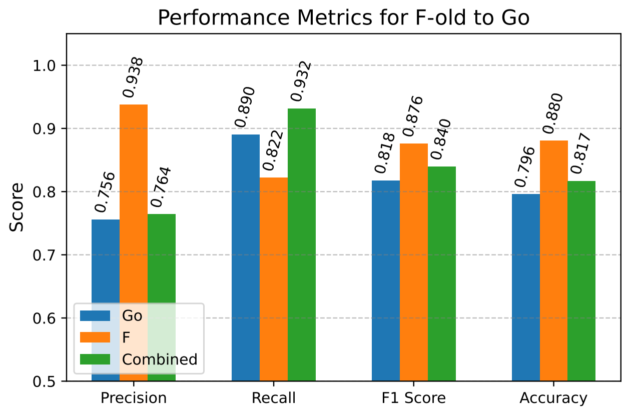

Figure 2: Data race detection performance for F-old programs translated to Go. We compare DRB's `libomp` based checker, Go's inbuilt Data Race Detector and a combined method. By using results from both Go's detector and TSan, the combined approach achieves highest recall at the cost of reduced precision.

maintaining high precision, indicating that the combined tools capture more true positives than either tool can alone.

Similar benefits are observed for **Fortran programs**. As seen in Figure 2 and Figure 3, for both the **F-old** and **F-new** sets, the combined approach achieves the highest recall among all tested configurations, in both cases at the cost of slightly lower precision.

5 Categorizing Translation Accuracy

Most programs appear to be translated faithfully, for example as shown in Listings 1 and 2. We focus in this section on selected programs for which our test-then-check approach resulted in false

Table 1: Evaluation metrics used. Tool Result is True iff a data race is detected.

Tool Result	Ground Truth		Recall(R)	Specificity	Precision(P)	Accuracy	F1 Score
	True	False					
True	TP	FP	$\frac{TP}{TP+FN}$	$\frac{TN}{TN+FP}$	$\frac{TP}{TP+FP}$	$\frac{TP+TN}{TP+FP+TN+FN}$	$\frac{2\cdot(P\cdot R)}{P+R}$
False	FN	TN					

Table 2: Benchmark Results(evaluated on DataRaceBench[13, 14] configurations as described in Section 4.1)

Tool	TP	TN	FP	FN	Accuracy	Precision	Specificity	Recall	F1 Score
C/C++	87	100	4	17	0.899	0.956	0.961	0.837	0.892
Go (translated C)	95	98	6	9	0.928	0.941	0.942	0.913	0.926
Combined (Go ∧ C/C++)	100	94	10	4	0.933	0.909	0.904	0.962	**0.935**
F-old	60	65	4	13	0.880	0.938	0.942	0.822	**0.876**
Go (translated F-old)	65	48	21	8	0.796	0.756	0.696	0.890	0.818
Combined(Go ∧ F-old)	68	48	21	5	0.817	0.764	0.696	0.932	0.840
F-new	58	57	5	13	0.865	0.921	0.919	0.817	**0.866**
Go (translated F-new)	63	45	17	8	0.812	0.787	0.726	0.887	0.834
Combined(Go ∧ F-new)	64	45	17	7	0.820	0.790	0.726	0.901	0.842

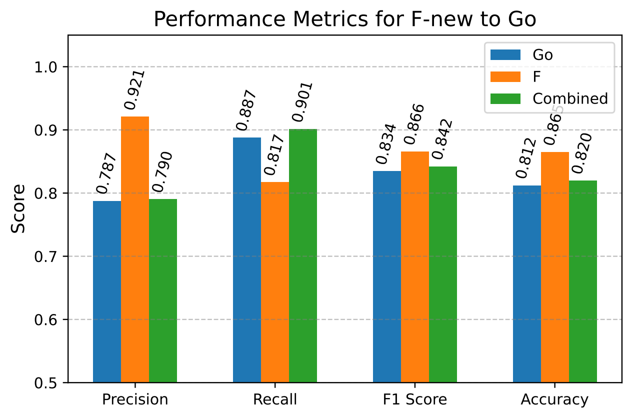

Figure 3: Data race detection performance for F-new programs translated to Go. We compare a flang + ThreadSanitizer based checker, Go's inbuilt Data Race Detector and a combined method. The combined method achieves an F1 score between the other approaches.

negatives or false positives, or where the result changed compared with TSan. We broadly categorize these cases in the following subsections. Note that comments are added for explanation in this paper, except when we explicitly refer to a comment as being generated by the AI agent during translation.

5.1 Translation enables easier detection

SIMD vectorization can be difficult to analyze for dynamic race detection tools. DataRaceBench contains 11 C programs with SIMD that have a data race (DRBs 24, 25, 115, 138, 157, 161, 164, 202, 204, 206, 207), for which TSan reports 7 false negatives, which is significantly below its overall accuracy across other DataRaceBench programs. Go does not provide explicit means of using SIMD, instead relying on the compiler to detect vectorizable loops. This prevents detection of a race in `DRB024-simdtruedep-orig-yes.c`.

```
#pragma omp simd
  for (i=0;i<len-1;i++)
    a[i+1]=a[i]+b[i]; // race
```

The Go translation instead uses a parallel loop with the same data race, which is detected. Note the green comments in the following code snippet, which were generated by Claude during translation, showing that the model was aware of the race in the source program code and intentionally created a race in the target program.

```
// Simulating SIMD parallelization with goroutines
// This creates the same data race pattern as the
// original SIMD code
var wg sync.WaitGroup
for i = 0; i < length-1; i++ {
  wg.Add(1)
  go func(index int) {
    defer wg.Done()
    a[index+1] = a[index] + b[index] // race
  }(i)
}
wg.Wait()
```

Surprisingly, the data race is eliminated in the Go translation of the equivalent Fortran program by using a sequential loop. Again, Claude documents this decision with a comment (shown in green) and apparently did not intend to reproduce a race in the target program.

```
// In Go, SIMD is handled by the compiler/runtime,
// we use regular loop
```

```
for i = 1; i <= length-1; i++ {
  // True dependence: a[i+1] depends on a[i]
  a[i] = a[i-1] + b[i-1] // no race, mistranslation
}
```

The difference in translation from Fortran and C are a recurring theme in our results. It is unclear to what extent this is simply caused by the non-deterministic nature of language models and could be overcome by repeated prompting for the same input programs, or whether differences in the amount of training data for C versus Fortran or the different nature of those languages play a role.

5.2 Translation introduces new race

Our translate-then-check approach leads to a total of 6 false positives, all of them caused by mistranslation that inserted an actual data race into the target program. This is interesting for two reasons. One, unlike TSan, the Go data race detector itself did not produce any false positives in our experiments, which is an indication of its maturity and tight integration with the language. Two, the AI agent introduces data races even for translation tasks in small, well-understood and previously seen programs such as those in DataRaceBench, at a rate that may be unacceptable in situations where the target programs are to be used on their own. On the other hand, the rate of introduced data races (6 out of 207) is not much higher than the rate of false positives reported by TSan on the original programs (4 out of 207), which means that AI translation may perform well enough for a test-then-check setting in practice.

The translated programs have data races for different reasons. For DRB078-taskdep2-orig-no and DRB107-taskgroup-orig-no, the Go programs correctly translate the bulk of the program, but use a sleep timer instead of an actual barrier in a flawed attempt to ensure that all goroutines finish their contribution before checking the final result. For the same programs, DRB078 and DRB107, the equivalent Fortran programs are translated correctly and use a WaitGroup for synchronization before the final check. For DRB107, the Fortran-to-Go translation instead introduces a new race by essentially making the loop counter a shared variable.

```
for threadID := 0; threadID < numCPU; threadID++ {
  go func() {
    if threadID == 0 { // race
      // ...
    }
  }() // should have passed threadID as argument here
}
```

This is because the loop counter is not passed as an argument to the spawned goroutine – unlike, for example, in Listing 2 – and instead becomes part of a closure. As a result, the loop in the main function and all spawned goroutines share the same instance of `threadID`, causing a race.

Other races are introduced because of OpenMP semantics that are not understood by the translator. For example, DRB127-tasking-threadprivate1-orig-no relies on the fact that two tasks either are scheduled on the same thread, meaning that they cannot run concurrently, or see different instances of a `threadprivate` variable, avoiding a data race. This subtle behavior is not captured correctly in the Go translation, where the tasks are implemented as goroutines that are not guaranteed to be scheduled on the same thread.

A slightly less subtle problem occurs in DRB174-non-sibling-taskdep-no, where the original program uses an OpenMP `single` construct to ensure two tasks run on the same thread and can therefore not race; the tasks are translated into Go routines, for which no such guarantees exist.

Perhaps among the most confusing cases is DRB129-mergeable-taskwait-orig-yes, a program that supposedly contains a race, but in fact can merely behave in two different ways depending on the implementation-defined choice of the OpenMP compiler and runtime whether or not to merge a task that has been declared as mergeable with its parent. Since the original program contains no actual data race, TSan is correct in reporting no race but gets flagged as false negative. The C-to-Go translation correctly reflects this behavior, and models the implementation-defined choice as a coin flip, where each option leads to a race-free execution. The Fortran-to-Go translation introduces a new data race that is unrelated to any of these subtle details. This race is detected in Go, leading to a somewhat misleading “true positive” designation.

5.3 Translation eliminates existing race

For some programs (such as the aforementioned Fortran version of DRB024-simdtruedep-orig-yes), the translation eliminates the existing data race. This is particularly worrying in a translate-then-check use case, because it increases the number of false negatives and can lead to false confidence in the original programs.

For example, the DRB037-truedepseconddimension-orig-yes program contains a nest of two loops. It would be safe to parallelize the outer loop, which is not parallelized. Instead, the inner loop is parallelized despite a true dependence, causing a detectable race in the original program. The C-to-Go translation removes the race by instead parallelizing the outer loop, leading to a false negative, whereas the F-to-Go translation maintains the original parallelization with data race, leading to a detected race.

Another interesting case, DRB142-acquirerelease-orig-yes, contains a data race because it uses acquire and release atomics incorrectly. Go does not have those weaker forms of atomic operations, and instead only offers sequentially consistent atomics. The original race thus cannot be accurately translated to Go.

Finally, DRB177-fib-taskdep-yes contains a data race while computing an expression that is stored in a variable that is never used again, presumably due to a typo. The translation eliminates this unnecessary expression and the race, and presumably a good optimizing compiler might do the same for the original C program.

```
#pragma omp task shared(i) depend(out : i)
  i = fib(n - 1);
#pragma omp task shared(j) depend(out : j)
  j = fib(n - 2);
#pragma omp task shared(i, j) depend(in : j)
  s = i + j; // race; i,j are used too early
#pragma omp taskwait
  return i + j; // s should have been returned
```

5.4 Detection differs between C and Go

For two programs, DRB013-nowait-orig-yes and DRB201-sync1-yes, the translation appears to accurately reflect the race in the original code, but the race detectors differ in their assessment. This is a testament to the non-deterministic nature of those tools. In the case of DRB013, TSan detects the race but Go does not. The original program is very simple:

```
#pragma omp parallel shared(b, error)
  {
#pragma omp for nowait
    for(i = 0; i < len; i++)
      a[i] = b + a[i]*5; // race

#pragma omp single
    error = a[9] + 1; // race
  }
```

The translated program has the same race, which is not detected unless the constant 9 is replaced with a larger number close to `len`.

DRB201-sync1-yes contains two memory accesses, one of which is not correctly guarded by a lock.

```
#pragma omp parallel num_threads(2)
  {
    if (tid == 0)
    {
      omp_set_lock(&l);
      x = 0; // race
      omp_unset_lock(&l);
    }
    else if (tid == 1)
    {
      omp_set_lock(&l);
      omp_unset_lock(&l);
      x = 1; // race, lock is no longer held
    }
```

The translation contains the same error, and Go successfully detects this in about 50% of our attempts.

6 Conclusion, Future Work

We investigated a translate-then-check approach that facilitates detection of data races in a source program by means of large-language-model (LLM)-based translation to a target language, followed by the use of conventional data race detection tools in the target language. Our experimental results, checking for races in DataRaceBench by means of translation to the Go programming language, show that translate-then-check can be competitive with state of the art data race detection tools applied to their native language. We argue that this approach is more reliable and explainable than pure LLM-based data race detection, and that translate-then-check presents an interesting use case of LLM-based language translation in which a small number of errors is more tolerable than in language translation tasks where the goal is to use the target programs in production.

It is worth stressing that our approach still relies on conventional data race detection tools in the target language, and we do not propose replacing conventional tools with AI agents, nor do we believe that our results allow a fair comparison between the race detectors in Go and Clang or Flang, partly due to the limitations of our experiments discussed earlier in the paper. Rather, the results indicate that existing data race detection tools can be useful beyond the languages and parallel frameworks for which they have been originally developed, with the help of AI-based translation.

A number of avenues remain unexplored and should be addressed in future work. We manually prompted the model to translate programs; it could be useful in practice to develop a data race detection tool that internally and transparently to the user queries a language model and applies a detector to the generated target program.

Our work does not analyze source code lines involved in data race pairs. While this is slightly more challenging due to the fact that races are detected in translated programs, it is conceivable that language models could be used to identify the original source lines, either by directly asking for this information, or by asking the translation model to include original source lines e.g. as comments in the translated programs.

Future work should also explore a wider range of AI models as well as prompts, for example by providing specific anti-patterns that are frequently used in the target language to avoid the introduction of new races during translation.

An even more interesting approach could be to involve an LLM-based AI agent directly in the detection task, by asking it to use the data race detection tool in the target language. If successful, this could lead to an iterative approach in which the agent improves the translation accuracy, and eventually fixes the data race in the original source program.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357. Support for this work was provided through the SciDAC program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, and the NSF-Simons AI Institute for the Sky (SkAI), 172 E. Chestnut St., Chicago, IL 60611, USA. We thank the anonymous reviewers for their valuable feedback.

References

[1] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H Ahn, Ignacio Laguna, Martin Schulz, Gregory L Lee, Joachim Protze, and Matthias S Müller. 2016. ARCHER: effectively spotting data races in large OpenMP applications. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 53–62.

[2] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 255–268. doi:10.1145/1806596.1806626

[3] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 211–230. doi:10.1145/582419.582440

[4] Nick Brown. 2025. Fully integrating the Flang Fortran compiler with standard MLIR. In *Proceedings of the SC '24 Workshops of the International Conference on*

High Performance Computing, Network, Storage, and Analysis (Atlanta, GA, USA) *(SC-W '24)*. IEEE Press, 939–949. doi:10.1109/SCW63240.2024.00133

[5] Milind Chabbi and Murali Krishna Ramanathan. 2022. A study of real-world data races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. ACM, 474–489. doi:10.1145/3519939.3523720

[6] Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei-Hung Lin, and Chunhua Liao. 2023. Data Race Detection Using Large Language Models. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 215–223. doi:10.1145/3624062.3624088

[7] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. doi:10.1145/945445.945468

[8] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. doi:10.1145/1542476.1542490

[9] Free Software Foundation, Inc. 2025. *GCC: The GNU Compiler Collection Reference Manual, Version 13.3.0*. GNU Project, Boston, MA. https://gcc.gnu.org/ Accessed 3 Aug 2025.

[10] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE109 (June 2025), 23 pages. doi:10.1145/3729379

[11] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. 2009. Helgrind+: An efficient dynamic race detector. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '09)*. IEEE Computer Society, USA, 1–13. doi:10.1109/IPDPS.2009.5160998

[12] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[13] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. doi:10.1145/3126908.3126958

[14] Pei-Hung Lin and Chunhua Liao. 2021. High-Precision Evaluation of Both Static and Dynamic Tools using DataRaceBench. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. 1–8. doi:10.1109/Correctness54621.2021.00011

[15] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: effective sampling for lightweight data-race detection. *SIGPLAN Not.* 44, 6 (June 2009), 134–143. doi:10.1145/1543135.1542491

[16] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. doi:10.1145/1133981.1134018

[17] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 167–178. doi:10.1145/781498.781528

[18] OpenMP Architecture Review Board (ARB). 2024. *OpenMP Application Programming Interface, Version 6.0*. OpenMP Architecture Review Board. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf

[19] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 866–866.

[20] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. doi:10.1145/265924.265927

[21] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. doi:10.1145/1791194.1791203