Sound Static Data Race Verification for C: Is the Race Lost?

KAROLIINE HOLTER, University of Tartu, Estonia
SIMMO SAAN, University of Tartu, Estonia
PATRICK LAM, University of Waterloo, Canada
VESAL VOJDANI, University of Tartu, Estonia

Sound static data race freedom verification has been a long-standing challenge in the field of programming languages. While actively researched a decade ago, most practical data race detection tools have since abandoned soundness. Is sound static race freedom verification for real-world C programs a lost cause? In this work, we investigate the obstacles to making significant progress in automated race freedom verification. We selected a benchmark suite of real-world programs and, as our primary contribution, extracted a set of coding idioms that represent fundamental barriers to verification. We expressed these idioms as micro-benchmarks and contributed them as evaluation tasks for the International Competition on Software Verification, SV-COMP. To understand the current state, we measure how sound automated verification tools competing in SV-COMP perform on these idioms and also when used out of the box on the real-world programs. For 8 of the 20 coding idioms, there does exist an automated race freedom verifier that can verify it; however, we also found significant unsoundness in leading verifiers, including Goblint and Deagle. Five of the 7 tools failed to return any result on any real-world benchmarks under our chosen resource limitations, with the remaining 2 tools verifying race freedom for 2 of the 18 programs and crashing or returning inconclusive results on the others. We thus show that state-of-the-art verifiers have both superficial and fundamental barriers to correctly analyzing real-world programs. These barriers constitute the open problems that must be solved to make progress on automated static data race freedom verification.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: Empirical Evaluation, Automated Software Verification, Concurrency, Data Race, Race Detection

ACM Reference Format:
Karoliine Holter, Simmo Saan, Patrick Lam, and Vesal Vojdani. 2025. Sound Static Data Race Verification for C: Is the Race Lost?. 1, 1 (April 2025), 41 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

It is important to ensure the reliability of C programs, as C is widely used in the development of critical infrastructural software. Vast amounts of legacy C code implementing these critical systems exist in the world. Concurrent C applications are vulnerable to data races, causing unpredictable behaviors, performance issues, and system failures. While dynamic and heuristic data race detectors are helpful, race-related vulnerabilities still continuously surface and are routinely patched in released software, showing the limits of unsound approaches. The ideal solution would be to verify the absence of data races at compile-time, using *sound* static data race analysis, throughout a

Authors' Contact Information: Karoliine Holter, University of Tartu, Tartu, Estonia, karoliine.holter@ut.ee; Simmo Saan, University of Tartu, Tartu, Estonia, simmo.saan@ut.ee; Patrick Lam, University of Waterloo, Waterloo, ON, Canada, patrick.lam@uwaterloo.ca; Vesal Vojdani, University of Tartu, Tartu, Estonia, vesal.vojdani@ut.ee.

ACM XXXX-XXXX/2025/4-ART
https://doi.org/XXXXXXX.XXXXXXX

program's evolution. Given the potential impact of sound static race detection for C programs, one may wonder what are the remaining problems that prevent its application in practice.

We have worked on sound static analysis of concurrent C programs for almost two decades, and implemented many of our ideas in the Goblint analyzer [88]. As an actively maintained open-source tool, it has found some practical use [86], but we have so far had limited success in verifying race freedom in real-world programs. We do not believe sound static race freedom verification is a lost cause, but we have reached a point where we are looking to re-calibrate and to explore some fresh ideas. We strongly believe that it is useful to know up-front what kind of concurrency schemes—specifically, schemes that are used in practice—pose problems for current tools.

More generally, we believe far more effort is needed to improve the state of benchmarking. Alglave et al. [3] identify the lack of standardized benchmarks as among the problems that prevent the development of robust and scalable tools and lament that the over-emphasis on novelty in research makes it difficult to develop tools beyond minimal prototypes. Rizzi et al. [68] substantiate their concern with some empirical evidence. They showed that making rudimentary improvements to an existing tool, KLEE [16], would have been more beneficial than many of the published new methods that compared to the original version of KLEE. They conclude that unguided innovation "may lead to wasted effort, missed opportunities for progress, an accretion of artifact complexity, and questionable research conclusions."

In response to such critiques and to incentivize the continuous improvement of tools, the community has established and participates in verification contests. Effectively organized verification contests can play a crucial role in developing a validated benchmark suite and ensuring reproducible evaluations. The Competition on Software Verification (SV-COMP) [9] is the pre-eminent verification contest for static analyzers. It incorporates community-based benchmarking processes that contribute to fairness and internal validity: tool developers may submit benchmarks, which are accepted as long as they are written in valid C. However, the tension between internal and external validity is a consideration for these contests, as outlined by Siegmund et al. [80]. To make comparisons meaningful, but at the cost of external validity, SV-COMP requires that benchmarks contain at most one property violation and that the ground truth be known; real programs are not subject to these constraints. While competition problems have helped drive the design and implementation of novel race detection techniques, one of our main goals is to further direct focus toward the automated verification of real-world programs. We are especially interested in understanding the gap between what is covered in today's competition problems and what is required to handle real-world programs.

In this paper, we identify obstacles to making significant progress on automated race freedom verification, in the context of software found "in the wild" today. We do so using the following three research questions.

(1) What are some concrete multithreaded implementation idioms preventing existing verifiers from verifying data race freedom in real-world programs?
(2) How well do existing verifiers handle each of these idioms when extracted into isolated micro-kernels?
(3) How prevalent are these challenging implementation idioms in real-world programs?

To identify the obstacles, we begin by evaluating how today's state-of-the-art automated data race verifiers perform on real-world programs out of the box. Having confirmed that automated verifiers cannot determine race freedom without manual intervention, we aim to understand the underlying causes of these failures. We extract unsolved data race verification challenges from the programs as idioms (RQ 1), formalize their underlying properties, and create micro-benchmarks of

the idioms to validate that they are yet unsolved (RQ 2) and prevalent in our selection of real-world programs (RQ 3).

Novelty. The main value of this paper is that we share, as co-developers of the top-performing data race verifier at SV-COMP, our insights into the limitations of these tools. Our study provides the following novel contributions:

- Clear empirical evidence that sound static analysis of data races is far from a solved problem.
- An in-depth analysis, using expert knowledge, of the fundamental causes of verification failures on real-world benchmarks.
- A set of idioms, collected from real-world benchmarks, that current state-of-the-art static race freedom verifiers have difficulty with; including

(1) a new set of micro-benchmarks that embody these idioms, and
(2) an evaluation of state-of-the-art verifiers on these micro-benchmarks.

Significance and Potential Impact. We have contributed our suite of micro-benchmarks to SV-COMP, and this paper explains our data-driven approach to deriving these micro-benchmarks. This kind of calibration research is important: by improving the quality of our benchmarking, we can, as a community, focus on problems that take us closer to the dream of static data race freedom verification of real-world programs. Our overarching vision is for there to exist techniques, and implementations thereof, that can successfully verify data race freedom for arbitrary real-world C programs.

Scope and Limitations. Our empirical study of real-world programs is limited to whole programs containing less than 500 KB of POSIX-threaded C code; specifically, these programs rely on the pthreads locking API for synchronization. Thus, we do not consider operating system components, such as device drivers, or other embedded code that rely on more fine-grained concurrency with interrupts and signal handling. Based on our previous experience with the analysis of device drivers [88], we note that many of the idioms identified in this study are also present in device drivers. However, the fine-grained and asynchronous nature of device driver code introduces additional challenges that are worth investigating separately.

2 Background

In this section, we recall definitions from automated software verification and clarify its role in the context of race detection. We also review existing efforts in benchmarking data race freedom verification methods, underlining current limitations and drawing attention to challenges that may hinder progress in the field.

2.1 Automated Software Verification for Data Races

In this paper, we focus on automated software verification, where the goal is to either prove that a program satisfies a property or produce evidence that the property is violated. Given a race verification task, an *automated data race freedom verifier* can respond with one of the following verdicts: True, meaning the program is free from data races; False, meaning the program definitely contains a data race; and Unknown, meaning the verifier can neither confirm nor deny the existence of a data race. A verifier is sound if it never responds True for any program containing a data race (or, equivalently, it has no false negatives). An unsound race detection tool may produce false negatives—it declares some programs that have data races to be race-free.

When considering larger programs, a static analyzer can provide more granular information than a single True/False/Unknown verdict, and produce a list of potentially racy accesses across the program (since programs may contain more than one race). It is, in principle, possible to manually

verify these as false positives and conclude that a program is guaranteed to be free of data races (i.e., that version of the program is verified); however, the necessity of repeating this manual step throughout a program's evolution limits the usability of a tool, especially so if the number of false positives is excessive. Expert users can fine-tune tools and annotate programs to prove the desired property, as described by Delmas and Souyris [22]; however, even in safety-critical settings, Nyberg et al. [57] identify automated verification as a key enabler and lack of tool expertise as a key obstacle to the adoption of formal verification.

Our goal is to assess the progress toward achieving automated verifiers capable of proving the *absence* of race conditions in real-world programs, and to point the way toward attaining this goal. We focus on understanding whether Unknown verdicts stem from trivial reasons, and if not, identifying some *unsolved verification challenges* causing the analyzer's automated reasoning to fail.

2.2 Benchmarks

One of the critical aspects influencing the progress of static data race analysis approaches lies in the available benchmarks. For many research problems, standardized benchmarks are readily available, such as DataRaceBench [50] for OpenMP. However, there are no standardized benchmark suites tailored for evaluating verifiers that address data races or other concurrency safety properties of C programs. Most of the recent benchmarking of concurrency analyses for C has been conducted using various editions of the SV-COMP benchmark set [10, 18, 27, 30, 36]. This set originates from the Competition on Software Verification, where fully automatic software verifiers for C and Java programs are evaluated. In the SV-COMP benchmark set, each verification task comprises a program and a specified property to assess, such as reachability, memory safety, overflows, termination, etc.

The SV-COMP data race category, introduced in SV-COMP 2022 [7], holds a variety of tasks for verifying the data race freedom property. The NoDataRace category, a subset of ConcurrencySafety, includes 783 verification tasks. According to SV-COMP 2023 results [8], 705 tasks have a True verdict (no data race), and 78 have a False verdict (a data race exists). Out of the 705 tasks with a True verdict, the tools in the competition altogether successfully verified data race freedom for 678 tasks, leaving only 27 tasks where the tools either generated false positives, encountered errors, or timed out. These results show that, over the SV-COMP NoDataRace benchmark set, solved verification tasks significantly outweigh unsolved ones. Even though the competing tools have solved many verification tasks, these tools have not yet been adopted for practical analysis of real-world programs. Thus, the competition results cannot be used to imply that almost all verification problems are solved—either in principle (on micro-benchmarks; as we show, important idioms are still missing) or in practice (on real-world programs, on which most tools do not run successfully). They do, however, demonstrate the range of problems that are within the reach of existing tools.

Although some verification tasks in the SV-COMP benchmark set remain unsolved, there is no basis to believe that the unsolved SV-COMP tasks fully capture the range of challenges faced when analyzing real-world programs. Conversely, we also cannot be sure about how often the idioms in these unsolved tasks occur in practice. Our claim is thus that existing benchmark sets do not provide a sufficient foundation for advancing data race detection. Our main goal in this work is to identify the obstacles hindering significant progress in automated data race verification and to distill these obstacles into new tasks that we contribute to benchmarks. Our work thus does the important work of enhancing the current benchmark sets to better reflect today's critical challenges, informed by a suite of real-world programs.

2.3 Defining Data Races in C: An Automated Verifier's Perspective

The C standard [44] gives an operational semantics for C programs and specifies the concurrent behavior in terms of a *happens-before* relation between memory accesses. When formalizing C/C++ memory models [5], the program's semantics (the set of possible executions) is defined by first considering the set of *pre-executions* admitted by the operational semantics of the language. If there is a pre-execution with a data race, the program's behavior is undefined; otherwise, the pre-executions are the executions of the program.

A pre-execution has a data race if it contains two accesses a and b to the same memory location by different threads, at least one of the accesses is a write, at least one of the accesses is non-atomic, and there is no happens-before relationship between them:

$$\begin{aligned} \mathit{data\_race}(a,b) \equiv\ & \mathit{mem}(a) = \mathit{mem}(b) \land \mathit{thread}(a) \neq \mathit{thread}(b) \\ & \land (\mathit{is\_write}(a) \lor \mathit{is\_write}(b)) \\ & \land (\neg \mathit{is\_atomic}(a) \lor \neg \mathit{is\_atomic}(b)) \\ & \land \neg(a \rightarrow_{hb} b \lor b \rightarrow_{hb} a). \end{aligned}$$

As races constitute undefined behavior, the automated verifier's aim is to prove their complete absence; in particular, so-called "benign races" should still be reported. The Data-Race Freedom (DRF) guarantee [14] states that if a program is data-race free under sequential consistency, then it exhibits only sequentially consistent behavior even if executed on weaker memory models.

A pair of accesses do not race if they are separated either in *space* (i.e., the memory locations are different: $\mathit{mem}(a) \neq \mathit{mem}(b)$) or in *time* (i.e., the accesses are ordered: $a \rightarrow_{hb} b \lor b \rightarrow_{hb} a$). Data race analysis is particularly challenging due to the interplay between the two dimensions of separation—in space and time—and reasoning about the two dimensions at once adds even more complexity to the analysis task. Thus, to devise scalable verification methods, it helps to focus on a subset of access pairs that can be shown safe through simpler reasoning.

This is how lock-based synchronization has been dealt with: instead of tracking the happens-before relations between the unlocks and locks of the same mutex, it suffices to track the set of locks $\mathit{locks}(a)$ held during an access a. When verifying code using lock-based synchronization, one can thus narrow the focus to accesses satisfying the condition

$$\mathit{mem}(a) = \mathit{mem}(b) \implies \mathit{locks}(a) \cap \mathit{locks}(b) \neq \emptyset,$$

meaning that if two accesses target the same memory location, they must share at least one common lock. An insightful observation by Naik and Aiken [56] is that such conditional formulations can simplify reasoning about heap-based locking idioms.

As a concrete example, consider the case of per-element locking [56, 62, 87], where a mutex within a struct protects a data field in the same struct. The code snippet on the right shows how the struct is accessed. For accesses a and b in an execution where two different threads call function `foo`, let s_a and s_b denote the addresses of the structs passed to the function. The above implication can then be proven as follows:

```
void foo(struct *s) {
  pthread_mutex_lock(&s->lock);
  s->data++;
  pthread_mutex_unlock(&s->lock);
}
```

$$\&(s_a{\rightarrow}\texttt{data}) = \&(s_b{\rightarrow}\texttt{data}) \implies s_a = s_b \implies s_a{\rightarrow}\texttt{lock} = s_b{\rightarrow}\texttt{lock}.$$

To verify that no access within `foo` races with any other access of `foo`, we can show that each access a individually satisfies condition $\forall s : \mathit{mem}(a) = \&(s{\rightarrow}\texttt{data}) \implies s{\rightarrow}\texttt{lock} \in \mathit{locks}(a)$, i.e., every access to the data field of a given s is protected by the corresponding lock of s. This enables more scalable local reasoning that rules out races for a subset of access pairs. We next identify non-lock-based idioms that can analogously serve as targets for novel verification approaches.

3 Challenging Idioms for Sound Static Data Race Analysis

We first present our most interesting results—the coding idioms themselves. Thus, this section will answer our first research question:

> **RQ 1.** What are some concrete multithreaded implementation idioms preventing existing verifiers from verifying data race freedom in real-world programs?

In Section 4, we explain the methodology we used to derive these idioms, and in Section 5, we validate their relevance and significance.

As highlighted in the previous section, we can achieve more scalable verification by targeting a subset of access pairs satisfying a simpler property that is sufficient but not necessary for showing that the data race property is not true. With this, we aim to capture the essential reason for certain accesses being safe. We thus call these the *essential properties* of access pairs. Often, the essential property can be proven for a set of access pairs by reasoning about the individual accesses involved, as in the example above where all accesses shared a common locking pattern.

We now present idioms that may appear to race but can be ruled out by a sufficiently sophisticated analysis. We classify these idioms based on whether the accesses are separated in space or time, characterize the essential properties they depend on, and suggest how these properties may be established. More complex idioms require joint reasoning about space and time to exclude races, and thus may rely on multiple essential properties. Where possible, we have factored such idioms into sub-idioms that can be considered independently. One challenge for sound static data race detection is that there are many ways to combine these idioms. We include some combinations in our benchmark suite, and our distillation of the idioms makes it possible for researchers to combine them in novel ways.

It is common for real-world applications to use an unbounded number of threads, as we show in Section 5.2. Therefore, all examples below spawn a non-deterministic number `n` copies of the thread `foo`. For space reasons, we omit all boilerplate and irrelevant details in the following figures. However, the artifact supplementing this paper (Data-Availability Statement) provides the idioms as micro-benchmarks and includes combinations of these idioms, as well as additional compositions not directly observed in the real-world programs we studied. We have checked our suite manually and with ThreadSanitizer [78, 79] to exclude potential undesired races. Our suite was peer-reviewed and accepted into the SV-COMP benchmark suite by the community.[1]

3.1 Space-Separated Accesses

In the category of space-separated accesses, we distinguish two particularly prominent subcategories: per-thread data and thread-local storage. Before conducting this analysis, we conjectured that the main obstacles were related to heap-allocated data, such as manipulating linked lists. However, we discovered that the major subcategory of not-yet-verifiable space-separated accesses turns out to be per-thread data—threads split work on a shared data structure, but each thread only accesses its own portion of the structure. Thus, for this family of idioms, the essential property is that accesses to the same memory location are only performed by the same thread:

$$mem(a) = mem(b) \implies thread(a) = thread(b).$$

This in turn can be proven by establishing a one-to-one relationship between the access locations and the identifiers of the accessing threads. In Section 6.2, we propose paths that future tools may use to establish such one-to-one relationships. In any case, there are variations on how the relation between threads and their data are maintained in practice, as we discuss below.

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1461

```c
int main() { // ...
  for (int i = 0; i < n; i++) {
    struct bar *b = malloc(sizeof(struct bar));
    pthread_create(&tids[i], NULL, &foo, b);
  }
  // ...
}
```

(a) per-thread-struct.

```c
int main() { // ...
  int *bs = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
    pthread_create(&tids[i], NULL, &foo, &bs[i]);
  // ...
}
```

(b) per-thread-array-ptr.

```c
int *bs;

int main() { // ...
  bs = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
    pthread_create(&tids[i], NULL, &foo, (void*)i);
  // ...
}
```

(c) per-thread-array-index.

```c
int main() { // ...
  int *bs = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++) {
    bs[i] = rand();
    pthread_create(&tids[i], NULL, &foo, &bs[i]);
  }
  // ...
}
```

(d) per-thread-array-init.

```c
int *bs;
int next_j = 0;
pthread_mutex_t next_j_mutex;

void *foo(void *arg) {
  pthread_mutex_lock(&next_j_mutex);
  int j = next_j++;
  pthread_mutex_unlock(&next_j_mutex);
  // use bs[j] ...
}

int main() { // ...
  bs = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
    pthread_create(&tids[i], NULL, &foo, NULL);
  // ...
}
```

(e) per-thread-index-inc.

Fig. 1. Paradigmatic examples of per-thread data idioms involving space-separation.

Per-thread data. Figure 1a illustrates a struct-based idiom where each spawned thread is provided a freshly allocated struct instance b via its thread argument. As long as these structs are not shared between threads, each thread can safely access its own instance. In Figure 1b, the main thread allocates an array of integers and passes distinct element pointers to all created threads. Figure 1c shows a variation where the array is global, and each thread is given an index into the array. Each thread is supposed to only access the array at its own index, though nothing (apart from developers wielding verification tools) prevents it from accessing other indices and thus causing races.

Often, the struct-based and array-based idioms are used together via an array of structs. Such idioms are regularly combined with thread joining: after all the child threads have completed their work, the parent thread can safely read results from the array. Combining with separation in time, the idiom shown in Figure 1d involves the main thread initializing the array at index `i`, then spawning the corresponding thread and passing ownership to that thread.

In more complex cases, the correspondence between threads and array indices is not fixed. Figure 1e shows each thread picking a unique index `j` for accessing the global array. Due to non-deterministic scheduling, the thread created in the i-th iteration of the loop may pick an index $j \neq i$. A variation of this idiom (not shown) uses a bitmask to record which indices have already been picked. Such idioms must ensure that different threads cannot pick the same index at the same time. In the bitmask version, threads may also unpick their index once they are done, allowing another thread to pick the same index but at a different time.

We now more rigorously characterize the idioms shown in Figure 1 by describing their syntactic and semantic properties, which highlight their defining features and similarities. In the specifications that follow, the parenthesized numbers (e.g. (1)) refer to the relevant syntactic properties, which are later used to assess how frequently the idioms occur in the benchmark set. The *emphasized text* highlights the syntactic differences among the idioms within the group.

Per-thread-struct (1a)

Syntactic: Threads are created within a looping construct (1), and a *freshly allocated struct* is given to created threads (2).
Semantic: The essential property holds as $mem(a)$ is freshly allocated for $thread(a)$.

Per-thread-array-ptr (1b)

Syntactic: Parent thread allocates a local array of data (bs), creates threads within a looping construct (1), and gives an *indexed address from the array* to each created thread (3).
Semantic: The essential property holds due to thread $t_i = thread(a)$, created in the i-th iteration of the loop, having exclusive access to $mem(a) = \&\texttt{bs}[i]$.

Per-thread-array-index (1c)

Syntactic: Parent thread performs the following steps: it allocates a *global* array of data (bs), creates threads in a loop (1), and assigns each thread a unique *index* to access the array bs (3).
Semantic: As with 1b: thread t_i has exclusive access to $mem(a) = \&\texttt{bs}[i]$. However, the way $thread(a)$ is mapped to $mem(a)$ differs in this case.

Per-thread-array-init (1d)

Syntactic: The parent thread allocates an array of data bs, and, within a looping construct (1), *initializes the array at index i with datum* $\texttt{bs}[i]$ before giving the indexed address from the array to the thread created at the i-th iteration (3).
Semantic: Here, there are two essential properties. The first essential property is that accesses by the spawned threads are separated in space: similar to both 1b and 1c, the thread t_i has exclusive access to $mem(a) = \&\texttt{bs}[i]$. However, before transferring ownership of $\texttt{bs}[i]$ to thread t_i, the main thread performs a write b to $\texttt{bs}[i]$. Thus, secondly, accesses are separated in time: the main thread's write b to $\texttt{bs}[i]$ occurs before thread t_i reads $\texttt{bs}[i]$ (access a), i.e., the second essential property is $b \rightarrow_{hb} a$ due to the synchronizing effect of thread creation.

Per-thread-index-inc (1e)

Syntactic: Threads are created within a looping construct (1). There is a globally accessible array of data (bs). *Global thread counter* next_j *is maintained by each spawned thread i* to obtain its unique ID j indicating its owned index of bs. Thus, a relationship is established between threads and array indices, ensuring disjoint accesses.
Semantic: The essential property is ensured by each $thread(a)$ owning a unique $mem(a) = \&\texttt{bs}[j]$ determined by the lock-protected counter next_j.

```c
__thread int b = 0;

void *foo(void *arg) {
  assert(b == 0);
  b = 1;
  assert(b == 1);
  // ...
}
```

(a) thread-local-value.

```c
pthread_key_t key; // initialized in main

void *foo(void *arg) {
  int x, y;
  pthread_setspecific(key, &x);
  assert(pthread_getspecific(key) == &x);
  pthread_setspecific(key, &y);
  assert(pthread_getspecific(key) == &y);
  // ...
}
```

(b) thread-local-pthread-value.

Fig. 2. Paradigmatic examples of thread-locality idioms involving space-separation.

Thread-local. Various forms of thread-local storage (TLS) exist that ensure space separation by construction. Figure 2a shows variable `b` declared in the global scope but using the `__thread` specifier provided by GCC. This specifier makes the variable thread-local instead of global, allowing each thread to access its own copy safely. For this reason, analyzers should not report races on `b`. C11 standardizes a similar specifier `_Thread_local` [45]. Although declared in the global scope, it may be necessary to analyze the values of `b` in a thread-local manner to verify the assertions. Moreover, pointers (either to local variables or dynamically allocated memory) assigned to such pointer-typed thread-local variables should not be considered to have escaped the thread.

Thread-local-value (2a)	Syntactic: A global variable is declared with the `__thread` specifier (4). Semantic: Ensured by construction.

TLS is also possible via the pthread library, as shown in Figure 2b, but only with pointer values. This idiom involves additional indirection whereby the thread-local variable is identified using an opaque pthread key. In one case, we observed such TLS being used to store a `setjmp`/`longjmp` buffer for thread-local exception handling.

Thread-local-pthread-value (2b)	Syntactic: A global variable is declared with the `pthread_key_t` type (5). Semantic: Ensured by construction.

3.2 Time-Separated Accesses

Because much work on static race detection has traditionally focused on lock-based synchronization (see Section 7), most race detectors handle locking fairly well—Section G (indirectly) shows that existing race detectors do not fail on straightforward locking. However, time separation does not require the use of locking—in principle, it only requires that accesses may not happen in parallel. We observed in our benchmarks that it is important to establish time separation for accesses that are not protected by locks but are otherwise separated by time (e.g., access a does not race with access b because $a \rightarrow_{hb} b$). The most common obstacle here is proving that threads have terminated.

```c
int main() { // ...
  for (int i = 0; i < n; i++)
    pthread_create(&tids[i], NULL, &foo, NULL);
  for (int i = 0; i < n; i++)
    pthread_join(tids[i], NULL);
  // ...
}
```

(a) thread-join-array-dynamic.

```c
int alive = 0;
pthread_mutex_t alive_mutex;

void *foo(void *arg) { // ...
  pthread_mutex_lock(&alive_mutex);
  alive--;
  pthread_mutex_unlock(&alive_mutex);
  return NULL;
}
```

```c
int main() { // ...
  for (int i = 0; i < n; i++) {
    pthread_mutex_lock(&alive_mutex);
    alive++;
    pthread_mutex_unlock(&alive_mutex);
    pthread_t tid;
    pthread_create(&tid, NULL, &foo, NULL);
  }
  // wait for all threads to stop
  pthread_mutex_lock(&alive_mutex);
  while (alive)
    // unlock and relock alive_mutex
  pthread_mutex_unlock(&alive_mutex);
  // ...
}
```

(b) thread-join-counter-outer.

Fig. 3. Paradigmatic examples of thread joining idioms involving time-separation.

Thread joining. As mentioned above, array-based per-thread data is used to return results from threads. For the main thread to safely access the results, it must ensure that all the involved threads have been joined. The most common idiom is to linearly join all created threads by their IDs, as shown in Figure 3a. However, the correctness of this obvious technique relies on many factors, e.g., the `tids` array must not be modified between the creating and joining loops, and the joining loop must exhaustively iterate over all the elements in the `tids` array.

Thread-join-array-dynamic (3a)

Syntactic: Threads are created within a looping construct (1). The main thread joins threads with array indexing (6).
Semantic: The essential property for each access a by threads created in the i-th iteration of the loop and accesses b by the main thread is the happens-before relation $a \rightarrow_{hb} b$ induced by thread joins.

A different idiom does not explicitly join threads but maintains a count of threads alive, as shown in Figure 3b. Once the count becomes 0, the main thread can safely access the results of the threads. This idiom heavily relies on the counting implementation being correct. A variation of this idiom (not shown) increases `alive` at the beginning of `foo` instead of the main thread. However, this requires an additional safeguard to prevent the main thread from continuing before any of the created threads have been scheduled to even increment the counter.

Thread-join-counter-outer (3b)

Syntactic: Threads are created within a looping construct (1). The thread creator has control dependency on a shared integer (7).
Semantic: The essential property is the same as for 3a, but the happens-before relation is enforced by the shared integer.

In one case (also not shown), we observed a hierarchical thread-joining scheme in the spirit of binomial heaps. Thread 0 joins threads 1, 2, 4, 8, 16, ...; thread 2 joins thread 3; thread 4 joins threads 5 and 6; thread 6 joins thread 7; etc. Because the scheme was implemented using bitwise operations, it is particularly challenging to ensure no thread is forgotten. Such hierarchical schemes are common in parallel computation frameworks.

```c
int b;

void *foo(void *arg) { // ...
  __sync_fetch_and_add(&b, 1);
  // ...
}
```

(a) atomic-gcc.

```c
int b = 0;
sem_t b_sem; // initialized with value 1

void *foo(void *arg) { // ...
  sem_wait(&b_sem);
  b = rand();
  sem_post(&b_sem);
  // ...
}
```

(b) semaphore-posix.

```c
int b = 0;
bool ready = false;
pthread_mutex_t ready_mutex;

void *foo(void *arg) {
  pthread_mutex_lock(&ready_mutex);
  while (!ready)
    // unlock and relock ready_mutex
  pthread_mutex_unlock(&ready_mutex);
  int x = b;
  // ...
}

int main() { // ...
  for (int i = 0; i < n; i++)
    pthread_create(&tids[i], NULL, &foo, NULL);

  b = rand();
  pthread_mutex_lock(&ready_mutex);
  ready = true;
  pthread_mutex_unlock(&ready_mutex);
  // ...
}
```

(c) value-barrier.

Fig. 4. Paradigmatic examples of other idioms involving time-separation.

Other. A lock-free way to avoid data races is to use atomic variables/operations. Figure 4a shows the atomic increment of `b` using a GCC-provided function. Analyzers must handle such functions to avoid reporting the races that the functions prevent. The C11 standard provides similar atomic variables and functions [45], but its versions also allow weaker memory semantics. While dynamic tools have progressed to detecting races in programs with weak atomics [51], static verifiers still predominantly assume sequential consistency, relying on the DRF property which guarantees that (absent weak atomics) it suffices to search for races only in sequentially consistent executions [14].

Atomic-gcc (4a)	Syntactic: Use of the `__sync` or `__atomic` prefixed operations (8). Semantic: The built-in atomics are sequentially consistent, but C11 atomics may require supporting weak memory semantics.

Besides mutexes, mutual exclusion can also be ensured using semaphores. In Figure 4b, a POSIX semaphore with an initial value of 1 is used as a mutex. In principle, semaphores with values greater than 1 could be used for more complex behavior. However, we did not observe any in our study.

Semaphore-posix (4b)	Syntactic: Use of POSIX semaphores (9). Semantic: When the semaphore is binary, we can define the set of held semaphores $locks(a)$ and reduce this to lock-based methods to ensure $mem(a) = mem(b) \implies locks(a) \cap locks(b) \neq \emptyset$.

Figure 4c illustrates a program using value-based synchronization to avoid data races. Writes to `b` in the main thread precede it setting the `ready` flag to true (under sequential consistency, at least). The created threads wait for `ready` to become true before reading from `b`. This protocol establishes a happens-before relation between the write to `b` and its reads. Instead of busy waiting, real-world programs would use condition variables [83] to broadcast the readiness signal. However, due to spurious wakeups, programs cannot merely rely on the condition variable but must also evaluate a value-based condition. More involved synchronization protocols may encode some state machine using multiple flags.

Value-barrier (4c)	Syntactic: Thread creator has control dependency on a shared int (7), use of signaling to conditional variables (10). Semantic: The essential property is symmetric to 3b; here, we need $b \rightarrow_{hb} a$ between accesses b of the main thread and the accesses a by threads created in the i-th iteration. This is enforced by the shared integer `ready`.

Although the idioms we captured represent a foundational subset of synchronization scenarios, they do not fully represent challenges from *all* real-world multithreaded C programs, particularly those that use fine-grained concurrency or specialized synchronization patterns for signaling. In our selected benchmarks, we did not observe such patterns, which are likely more prevalent in specific domains such as operating system kernels and certain high-performance POSIX applications. With a sufficiently large set of general-purpose POSIX programs, we would eventually expect to find instances where synchronization techniques from these specialized domains have migrated to general-purpose POSIX programs. This would contribute new idioms under the *Other* category. We claim that the idioms that we have already identified, on our significant C corpus, form a good starting point for research today, but acknowledge that the set is not necessarily exhaustive.

4 Methodology

One of our goals was to identify multithreaded implementation idioms that prevent existing verifiers from confirming data race freedom in real-world programs. In the previous section (3), we outlined these challenges. In this section, we detail our experimental approach to achieving this

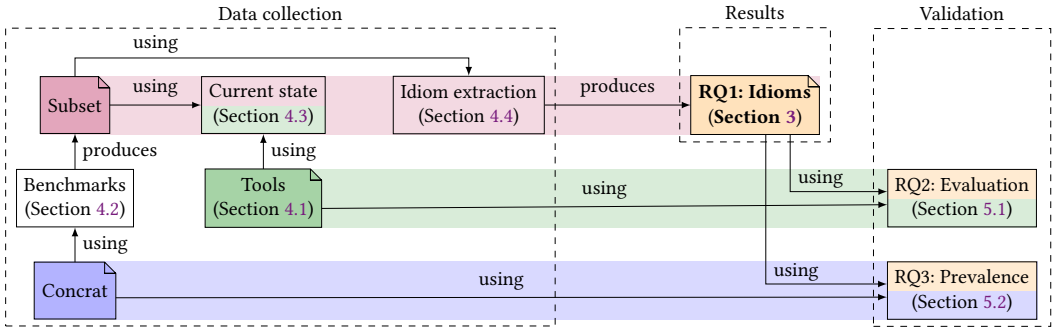

Fig. 5. Overview of our research design. We extract idioms from a Subset of the Concrat suite, then validate that the extracted idioms represent open problems by evaluating all tools on our micro-benchmarks, and we validate the relevance of the problems by counting the occurrences of the idioms in the entire Concrat suite.

goal, including the selection of automated race freedom verifiers, the choice of real-world programs, and the process of evaluating the results of the selected tools on these programs. Figure 5 illustrates how the data objects described in this section are used throughout the study, and which processes contribute to each part of the study, altogether providing an overview of our research design.

4.1 Tools Under Evaluation

Recall that in this study we are interested in *sound static* data race freedom verification. Thus, the automated verification tools must incorporate methods for verifying data race freedom, and be based on static analysis techniques which are conceptually sound, while recognizing that implementation flaws may introduce unsoundness.

First, we take under evaluation the tools from SV-COMP that participated in verifying tasks from the NoDataRace category in SV-COMP 2023 and achieved a positive score. A positive score in the scoring system of SV-COMP [8] indicates that the tool indeed successfully verified programs and, at the same time, did not make false claims about faulty programs being correct (i.e. they were sound), with few exceptions. The selected tools from SV-COMP are listed in the upper portion of Table 1, along with their results in the NoDataRace category of SV-COMP 2023. Most of these tools are actively being developed, and their detailed engineering statistics can be found in Section A. From the Ultimate tool family (Automizer, Taipan, GemCutter) we only take the best one (UAutomizer) as the results across the entire family are similar. We also assessed the tools from SV-COMP 2024 [9], with the results detailed in Section 6.1. The findings and conclusions of our present study are unchanged from SV-COMP 2023 to SV-COMP 2024.

We also experimented with automated verification tools from the literature that do not participate in SV-COMP (bottom part of Table 1). We excluded the Relay analyzer because we could not build a binary that would run reliably—the latest edits to the code date back to 2010. Even though we did manage to run Relay (unreliably), our experience did not give us confidence that its results will be independently reproducible across different machines. We excluded Coverity Scan due to license restrictions that prevent publishing its evaluation results. Nonetheless, our experiments indicated that including it would not have changed our conclusions: it does not show better results on our micro-benchmarks than the tools that we report on. Coverity Scan combines classical data flow analysis techniques with heuristics and statistical methods [26], and does not claim to be sound. Thus, we believe it is safe to reveal that, in our internal testing on the micro-benchmarks, it reported false negatives.

Table 1. List of automated verification tools that support data race freedom verification. Tools selected from SV-COMP 2023 are presented with their results in the NoDataRace category of SV-COMP 2023. Correctly and incorrectly solved verification task counts are given by expected verdict (True or False).

Verification tool	Ref.	Approach	Correct			Incorrect		Score	License[2]
			True	False[1]		True	False		
Goblint	[69, 88]	Abstract interpretation	652	0	(0)	0	0	1,304	✓
Deagle	[36, 37]	Satisfiability modulo theory (SMT)	588	51	(60)	0	1	1,211	✓
Dartagnan	[32, 60]	Bounded model checking (BMC)	464	0	(62)	2	6	768	✓
UAutomizer	[38, 39]		378	0	(59)	0	0	756	✓
UGemCutter	[28, 48]	Automata based model checking	366	0	(67)	0	0	732	✓
UTaipan	[24, 34]		314	0	(62)	0	1	612	✓
CPAchecker	[11, 21]	SMT-based model checking	200	0	(29)	0	0	400	✓
Locksmith	[62]	Constraint-based data-flow analysis	129	0	(0)	1	0	226	✓
Theta	[1, 84]	Abstraction-based model checking	98	9	(9)	0	0	205	✓
Relay	[90]	Symbolic interprocedural data-flow analysis	Does not participate in SV-COMP						✓
Coverity Scan	[25, 26]	Data-flow analysis and statistical deviance							✗

[1] In parentheses: number of unconfirmed (no witness provided) results. SV-COMP 2023 erroneously demanded witnesses for data races even though no witness format had been adopted by the community for data races: https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/merge_requests/398.

[2] Whether licence permits publishing evaluation results (✓ — permits, ✗ — does not permit).

4.2 Benchmark Selection

To find out why tools struggle with verifying real-world programs, we must identify a sufficiently large and interesting set of benchmarks. Note that a benchmark set can yield valuable results even without being representative. One large and interesting benchmark set found in recent literature is the Concrat benchmark set. Hong and Ryu [43] assembled the original suite to evaluate their Concrat tool, which aids in automated C to Rust translation of multithreaded programs (thus aligning well with our study of data race freedom verification tools). This suite consists of 46 C projects with public GitHub repositories satisfying the following criteria [43]:

- more than 1,000 stars,
- not study material,
- using the pthread lock API at least once,
- C code less than 500,000 bytes, and
- translatable with C2Rust.

The last condition resulted in the exclusion of two projects due to the use of C11 atomics because C2Rust supports only C99-compliant code. Based on early experiments with various tools, we observed and fixed syntactic issues in 15 of the benchmarks, presenting further details in Section B. We make our set of the fixed Concrat benchmarks available in the artifact (see Section 8).

A key difference between the SV-COMP suite and the Concrat suite is that SV-COMP programs must be complete, while Concrat programs can be incomplete. Tools targeting SV-COMP relied on the completeness assumption. To evaluate the tools on the Concrat suite, we were therefore forced to additionally exclude 28 programs from the fixed Concrat suite for the following reasons:

- 5 programs are incomplete because they contain no `main` function.
- 5 programs are single-threaded because they never call `pthread_create`.
- 1 program is incomplete because it uses significant *external* assembly code.
- 17 programs are incomplete because they have dependencies on libraries, such as OpenSSL, curl, and Lua, which are not part of the Concrat benchmark suite.

The excluded programs, along with the reasons for their exclusion, are listed in Section C, and the descriptions of the included programs in Section D. We use the remaining 18 programs for evaluation and call them the Subset. This subset is skewed towards programs that are not dependent on large external libraries. However, if our subset is a biased sample of the larger suite, the bias is in favor of simpler programs. We point out that if a state-of-the-art race freedom verifier can handle arbitrary real-world programs, it should also be able to handle these programs, or at the very least the concurrency idioms from them.

4.3 Mapping the Current State: Tools on Real-World Programs

It is reasonable to assume that state-of-the-art automated verification tools for concurrent C programs cannot yet verify real-world programs, as these tools compete on verifying the small, clean examples from SV-COMP; success on larger, messier programs would be unexpected. To validate the assumption, we run the selected seven verification tools on the Concrat benchmark Subset and evaluate how they perform on real-world programs out of the box. This approach aligns with our goal of assessing the tools' ability to operate automatically and identifying where they encounter difficulties without manual intervention. It is important to note that our overall objective was not to assess the performance of these tools on real-world programs but to investigate the obstacles to fully automated data race verification.

There are scenarios where users take a more active role by adding annotations, creating mocks or manually tuning tool configurations to better suit the program being analyzed to ensure the analysis succeeds. However, as our primary focus is on *automation*, we refrained from assisting the tools with practices that rely on manual intervention. The SV-COMP rules require tool developers to provide configurations optimized for general applicability with respect to the property being analyzed.

Thus, we use the tools with their SV-COMP 2023 configurations under increased resource limits compared to the competition's resource limits. Each run is limited to 2 hours of CPU time, 24 GB of RAM, and 8 CPU cores instead of 15 minutes of CPU time, 15 GB of RAM, and 8 CPU cores as in SV-COMP. To provide a comparison point, the Concrat benchmark Subset contains 18 programs with a mean size of 6,664 LoC, while the SV-COMP NoDataRace set contains 783 programs with a mean size of 180 LoC.

Table 2 presents the results of evaluating the selected SV-COMP tools on the Concrat benchmark Subset. Five tools only produce errors or exhaust resources on the suite of real-world programs:

(1) Deagle exhausts given resources without completing the analysis for half of the programs. On the other half, it mainly lacks support for some standard library functions.
(2) Dartagnan primarily fails due to missing standard library function support, e.g., `fprintf` and `time`[2]. In other cases, it does not support calling `pthread_create` in a loop.
(3) UAutomizer fails for various reasons, most often due to missing function support, e.g., `sin` and `pthread_rwlock_init`. In many instances, its front-end fails, either due to parsing or unsupported type casts.
(4) CPAchecker does not support various thread creation and thread joining patterns, as well as condition variables. Notably, it lacks the support for the `memset` function from the C standard library.
(5) Theta fails to parse all of our real-world programs, always due to `__attribute__` from standard library headers.

[2]Section F presents complete details on which functions are unsupported for Dartagnan, UAtomizer, and CPAchecker.

Table 2. Experimental results of the selected tools with SV-COMP 2023 settings on the included Subset of the Concrat benchmark set with 2 hours of CPU time, 24 GB of RAM, and 8 CPU cores. The second column indicates program size in lines of code (LoC).

Benchmark	LoC	Goblint	Deagle	Dartagnan	UAutomizer	CPAchecker	Locksmith	Theta
axel*	5,848	Timeout	LibFun	LibFun	Parsing	LibFun	Crash	Parsing
C-Thread-Pool*	710	Unknown	LibFun	LibFun	Unsupp	Unsupp	Unknown	Parsing
dnspod-sr	9,259	Timeout	LibFun	Unsupp	Unsupp	LibFun	Unknown	Parsing
EasyLogger*	2,011	Unknown	LibFun	LibFun	LibFun	Unsupp	Timeout	Parsing
fzy	2,621	Unknown	Timeout	LibFun	Timeout	Unsupp	Unknown	Parsing
klib*	716	Unknown	Timeout	LibFun	Unsupp	Unsupp	Unknown	Parsing
level-ip	5,414	Timeout	LibFun	Unsupp	LibFun	LibFun	Crash	Parsing
libfaketime	521	True	LibFun	LibFun	Crash	Unsupp	True	Parsing
lmdb	10,827	Crash	LibFun	LibFun	LibFun	LibFun	Timeout	Parsing
Mirai-Source-Code	1,839	Crash	Timeout	LibFun	Timeout	Unsupp	Unknown	Parsing
nnn	12,091	Unknown	OOM	LibFun	LibFun	Timeout	Unknown	Parsing
phpspy	19,390	Unknown	LibFun	Unsupp	Parsing	Unsupp	Timeout	Parsing
pigz	9,118	OOM	LibFun	Unsupp	LibFun	Timeout	Timeout	Parsing
ProcDump-for-Linux*	4,152	Unknown	Timeout	LibFun	Parsing	Unsupp	Unknown	Parsing
Remotery*	7,212	Crash	Timeout	LibFun	LibFun	Unsupp	Timeout	Parsing
streem*	20,169	Crash	Timeout	Timeout	LibFun	LibFun	Timeout	Parsing
the_silver_searcher*	7,242	Timeout	Parsing	Crash	Unsupp	LibFun	Unknown	Parsing
uthash	817	True	Timeout	LibFun	LibFun	Unsupp	Unknown	Parsing

True — Tool claims data race freedom
Unknown — Neither claimed verified nor refuted
OOM — Out of memory
Timeout — Not finished within time limit

LibFun — Unsupported library function
Unsupp — Unsupported syntax
Parsing — Parsing failed
Crash — Tool crashed

* ThreadSanitizer found a data race.

The remaining two tools give some results besides just errors:

(1) Locksmith finishes its analysis for ten programs, including verifying one to be race-free and producing data race warnings (i.e., results in Unknown) for the others. It crashes twice, reporting a "typing bug", and runs out of time in six cases.
(2) Goblint verifies two programs and finishes with data race warnings for seven. It crashes while analyzing four programs and runs out of time or resources on another five.

Notably, none of the static tools, including those employing under-approximating methods, reported a counter-example—specifically, a concrete race condition—in these real-world programs. We also ran ThreadSanitizer, further detailed in Section E, to dynamically assess whether the programs exhibit observable data races. ThreadSanitizer reported data races for 7 of the programs.

Summarizing the tool results on real-world programs, one barrier that we found for 6 of the 7 tools was scalability. While this could be overcome to some extent by adjusting precision through configurations, such changes can only stave off resource exhaustion and resolve a subset of Unknown verdicts. Other cases require fixes and improvements to the tools themselves, as described above. Given that we have observed that these tools struggle with programs that are already close but not entirely compliant with the tools' requirements, we conclude that they are not yet capable of analyzing real-world programs. While the non-core issues that we have outlined here (e.g., missing function support) may be fixed by the tool developers by applying more polish and elbow grease, our primary research interest is in the conceptual limitations of their verification approaches.

4.4 Identifying and extracting the challenging idioms

The main contribution of this paper is the identification and crystallization of idioms (Section 3) that are yet unhandled by existing automated data race verifiers. This section presents our approach to extracting the idioms, and details some additional checks that we performed for completeness.

Unsolved Data Race Verification Challenges from SV-COMP. We first analyzed (for completeness) the failures of the verifiers on the SV-COMP suite to gain a comprehensive understanding of the currently known idioms that the tools struggle to verify. The details of this analysis can be found in Section G. We considered the 11 tasks with a True verdict that *all* of the automated race freedom verifiers from Section 4.1 failed to prove correct at SV-COMP. However, these 11 unsolved tasks can be seen to be somewhat arbitrary (since SV-COMP comes from community contributions), and insufficient as a guide for advancing data race freedom verification. We thus systematically look for more unsolved race detection tasks in real-world programs. In doing so, we used the idioms from SV-COMP's existing unsolved tasks as a reference to ensure that the idioms we identified from real-world programs were new and previously undocumented.

Unsolved Data Race Verification Challenges in the Concrat Suite. To collect unsolved data race verification challenges from the Concrat Suite, we first used a verification tool on the Concrat Subset programs to generate warnings. These warnings served as a guide to identifying the coding idioms causing the verification to fail. This process involved inspecting the analysis warnings and reasoning backwards to their origins, where the most difficult part is understanding the analyzed programs themselves. To identify the idioms, we relied on our expert knowledge, concentrating on those idioms that pose the greatest barriers to verification. For instance, if thread joining idioms are not properly handled, many of the reported races on accesses following the join will be false alarms. Once the idiom was identified, we extracted a kernel that captured the verification challenge, validated that it is indeed unsolved by running the verification tools, and finally, created variations of these kernels to test the soundness of the verifiers.

For our manual analysis, we chose to use Goblint's warnings for several reasons. Among the tools evaluated, only Goblint and Locksmith were capable of processing some Concrat programs right out of the box. In contrast, other tools encountered only timeouts, unsupported features, or crashes, making them unfit for supporting the process of extracting new idioms. Unlike most verification tools at SV-COMP, Goblint performs thread-modular abstract interpretation [54, 73, 74], producing an exhaustive list of warnings with all races it failed to disprove. This makes it well-suited for aiding in identifying challenging idioms—even in programs with unknown ground truth—because it provides a comprehensive view of how automated reasoning fails. Our investigation is not dependent on the ground truth—whether a given access pair actually is a race—rather, we focus on features in the underlying program that causes such loss of precision to produce bad abstract states (and thus Unknown verdicts) for a significant portion of access pairs. Additionally, as Goblint is a tool we develop, it allowed us to use our expertise during analysis.

To begin with analyzing the warnings, two Concrat programs were verified by Goblint, so no further work was required. Of the remaining 16 that were not verified, the warnings from the 7 programs where Goblint returned an Unknown verdict were directly used as a guide to identify idioms. For further investigation of the 8 benchmarks on which Goblint did not result in warnings (i.e., reached resource limits or crashed), we used its provided configuration for analyzing larger programs instead of its SV-COMP configuration. The goal of using this configuration was not to improve verification chances but to avoid timeouts in favor of obtaining more warnings: it deactivates many resource-intensive analyses, trading precision for efficiency. This efficiency

Table 3. Coding idioms that resulted in false positive warnings by Goblint on the Concrat suite.

Benchmark	Identified causes for failure to verify
axel	— (not inspected, Goblint crashed)
C-Thread-Pool	Thread joining with counter incremented within the thread, per-thread struct
dnspod-sr	Thread joining (array), per-thread struct (in array), atomics
EasyLogger	Semaphore
fzy	Thread joining with binomial heap, per-thread struct (in array), thread-local heap memory
klib	Thread joining (array), per-thread struct (in array), atomics
level-ip	Thread joining (array), per-thread struct, thread-local heap memory
libfaketime	— (verified)
lmdb	Per-thread struct, thread-local storage (pthread)
Mirai-Source-Code	— (none identified)
nnn	Thread joining with counter incremented outside the thread, per-thread index (bitmask)
phpspy	Thread joining (array), per-thread array (by index), atomics (GCC)
pigz	Per-thread struct, thread-local variable (pthread)
ProcDump-for-Linux	Per-thread index (simple), per-thread struct (in array), semaphore
Remotery	Per-thread struct, thread-local storage (pthread), atomics (GCC)
streem	— (not inspected, Goblint timed out)
the_silver_searcher	Per-thread array pointers, thread-local variable (GCC)
uthash	— (verified)

allowed Goblint to generate warnings for all benchmarks save one, where it still timed out. To summarize Goblint results with the larger programs configuration on the 18 programs:

2 True, 13 Unknown (causes identified), 1 Unknown (unclear cause), 2 Crash/Timeout.

Table 3 summarizes the idioms identified to cause false positives in the 13 programs.

5 Validation

We validate the extracted idioms in two ways to confirm their significance and relevance. First, we evaluate all of our selected tools on the extracted micro-benchmarks—code snippets that directly correspond to the extracted idioms—to confirm that these idioms represent open problems in automated data race verification. Second, we examine the prevalence of these idioms by quantifying their occurrences across the entire Concrat benchmark suite.

5.1 Evaluation of the Race Freedom Verifiers on the Extracted Examples

After identifying the coding idioms that contributed to the false alarms in the real-world benchmarks, we wanted to ensure that the verification of the found idioms is indeed not solved by the tools. Thus, we extracted the idioms into micro-benchmarks to evaluate the selected tools on these idioms in isolation. Where possible, we also created racy variations of the race-free examples, which we use to determine the soundness of verification. We did not create such variations for thread-local variables, because making them non-thread-local would simply turn them into global variables, which would not pose a challenge for the tools.

RQ 2. How well do existing verifiers handle each of these idioms when extracted into isolated micro-kernels?

Table 4 shows the results of running the SV-COMP analyzers on the extracted micro-benchmarks. If a tool verifies a race-free benchmark while accurately distinguishing between correct and incorrect versions, it has soundly verified the example, and “Correct” will be indicated in the table. “Inconclusive” indicates that the tool cannot differentiate between the race-free and racy examples,

Table 4. Experimental combined results of the selected tools on race-free examples and their racy variations.

Extracted example	Fig.	Goblint	Deagle	Dartagnan	UAutomizer	CPAchecker	Locksmith	Theta
per-thread-struct	1a	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing
per-thread-array-ptr	1b	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing
per-thread-array-index	1c	Inconcl	Inconcl	Unsupp	Timeout	Unsupp	Inconcl	Parsing
per-thread-array-init	1d	Unsound	Inconcl	Unsupp	Timeout	Unsupp	Inconcl	Parsing
per-thread-struct-in-array	-	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing
per-thread-index-inc	1e	Inconcl	Inconcl	Unsupp	Timeout	Unsupp	Inconcl	Parsing
per-thread-index-bitmask	-	Inconcl	Timeout	Unsupp	Timeout	Unsupp	Inconcl	Parsing
thread-local-value	2a	Inconcl	Correct	Unsupp	Inconcl	Unsupp	Inconcl	Parsing
thread-local-value-cond*	-	Inconcl	Correct	Unsupp	Inconcl	Unsupp	Inconcl	Parsing
thread-local-value-dynamic*	-	Inconcl	Correct	Unsupp	Inconcl	Unsupp	Inconcl	Parsing
thread-local-pthread-value*	2b	Correct	LibFun	Unsupp	LibFun	Unsupp	Correct	Parsing
thread-local-pthread-value-cond*	-	Inconcl	LibFun	Unsupp	LibFun	Unsupp	Inconcl	Parsing
thread-join-array-dynamic	3a	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing
thread-join-array-const	-	Unsound	Unsound	Unsupp	Correct	Unsupp	Inconcl	Parsing
thread-join-binomial	-	Inconcl	Unsound	Unsupp	OOM	Unsupp	Inconcl	Parsing
thread-join-counter-outer	3b	Inconcl	Correct	Unsupp	OOM	Unsupp	Inconcl	Parsing
thread-join-counter-inner	-	Inconcl	Timeout	Unsupp	Timeout	Unsupp	Inconcl	Parsing
atomic-gcc*	4a	Inconcl	Correct	Unsupp	OOM	Unsupp	Correct	Parsing
semaphore-posix	4b	Inconcl	Inconcl	Unsupp	Inconcl	Unsupp	Unsound	Parsing
value-barrier	4c	Inconcl	Correct	Unsupp	OOM	Unsupp	Inconcl	Parsing

Correct — Soundly verified
Inconcl — Inconclusive
OOM — Out of memory
Timeout — Not finished within time limit
Unsound — Unsoundly verified
LibFun — Unsupported library function
Unsupp — Unsupported syntax
Parsing — Parsing failed
* Extracted example does not have racy variations.

yet it maintains soundness (i.e., it yields no false negatives). However, if a tool verifies a race-free example but incorrectly labels the racy variation as race-free, we indicate "Unsound," as it has not truly verified the idiom.

The winning tool at SV-COMP, Goblint, verifies a simple case with thread-local storage, but is unsound in two other cases, despite having no incorrect results at SV-COMP 2023. The second-ranking tool, Deagle, correctly handles 3 examples with thread-local storage and atomics. Furthermore, it soundly verifies one challenging example involving a threads counter, in addition to a simpler value-based barrier. Unfortunately, Deagle falsely claims data race freedom in six cases. UAutomizer can only verify an example where a constant number of threads are joined, while exhausting given resources on many other examples involving an unbounded number of threads. Locksmith soundly verifies some examples with thread-local storage from the pthread library and atomics, but treats the semaphores incorrectly. Other tools encounter similar failures on the examples as on the real-world programs. Dartagnan fails on all benchmarks due to its inability to handle thread creation within a loop, which is a common feature in each example. CPAchecker struggles because it does not support arrays of threads. Theta encounters parsing issues as it lacks support for struct and union definitions in the included stdlib.h and pthread.h headers.

Finding 2. Leading SV-COMP automated data race freedom verifiers are unable to reason about most thread-joining and per-thread data distribution implementation idioms that we have extracted from real-world programs, even when such idioms are isolated and simplified. At least one tool succeeds for 8 of the 20 idioms.

5.2 Prevalence of the Identified Idioms in the complete Concrat Suite

As the idioms were extracted by manually inspecting the results of the Goblint tool on a subset of the Concrat suite, we wanted to ensure that the identified idioms are not isolated cases. Thus, the quantitative question of how frequently these idioms occur in real-world programs is an orthogonal one. By construction, the idioms occur at least once on the Concrat suite. But do they occur significantly more than once? To substantiate the claim that these are prevalent issues, we consider the entire Concrat benchmark suite, and conduct a query-based analysis to systematically count the occurrence of these difficult idioms, thus obtaining more representative numbers summarizing their prevalence. We can then compare these numbers with the existing tasks in SV-COMP to confirm that our added benchmarks complement the existing benchmarks.

RQ 3. How prevalent are these challenging implementation idioms in real-world programs?

To search for and count the occurrences of these challenging implementation idioms in our benchmarks, we used Joern [91], relying on its control flow analysis. To characterize benchmarks' approach to per-thread data, we ran a query with the Joern analyzer to find all functions that can result in the creation of threads, including `pthread_create` itself and any functions that eventually invoke it. We counted any such functions that are called in a looping construct (1). Next, we computed whether there exists data flow between a memory allocation site and a thread creation site (2). For the array-based idioms, we considered whether an integer variable used in a loop or an address expression with indexing (or pointer addition) is given to a spawned thread (3). For the join idioms, we considered if there is array indexing involved (6). The syntactic feature relevant to both thread counters and value-barriers is the control-flow dependence of a thread creator on a shared variable (7); thus, we searched for the use of shared variables in conditions of control statements in a function spawning a thread. The rest of the features only require a keyword search in the source code; a comprehensive list is given in Section H.

The analysis outputs log files in human-readable Markdown format with the relevant information, including convenient links to the source code, which we inspected manually to validate that queries work as expected. Table 5 presents the results of the syntactic analysis for the prevalence of

Table 5. Syntactic properties indicative of the use of the identified patterns. Numbers (in # column) serve to identify specific idioms. Counts (in the three right-most columns) indicate how many benchmarks in each suite satisfy a given property. Concrat refers to the complete Concrat suite, while Subset is Concrat minus incomplete, single-threaded, and library-dependent programs.

Idioms	#	Relevant Property	SV-COMP'23		Concrat		Subset	
Per-Thread (Figure 1)	1	Thread creation within a looping construct	87	(11.1%)	29	(64.4%)	13	(72.2%)
	2	Flow from memory allocation to thread creation	6	(0.8%)	22	(48.9%)	9	(50.0%)
	3	Giving an integer or indexed address to created thread	4	(0.5%)	13	(28.9%)	9	(50.0%)
Thread-Loc. (Figure 2)	4	Use of thread-local storage (`__thread`)	0	(0.0%)	2	(4.4%)	1	(5.6%)
	5	Use of Posix API for thread-local data (`pthread_key`)	2	(0.3%)	4	(8.9%)	3	(16.7%)
Thread Join (Figure 3)	6	Joining threads with array indexing	54	(6.9%)	15	(33.3%)	8	(44.4%)
	7	Thread creator has control dependency on shared int	13	(1.7%)	10	(22.2%)	4	(22.2%)
Other Sync. (Figure 4)	8	Atomics (`__sync` or `__atomic`)	0	(0.0%)	8	(17.8%)	5	(27.8%)
	9	Semaphores (`sem_wait`)	0	(0.0%)	3	(6.7%)	2	(11.1%)
	10	Use of `pthread_cond`	7	(0.9%)	20	(44.4%)	9	(50.0%)
		Total	783	(100.0%)	45	(100.0%)	18	(100.0%)

the unsolved verification tasks in SV-COMP as well as Concrat (both the complete set and the Subset described in Section 4.2). The results correspond fairly well to the more detailed root cause analysis of Section 3 on the subset of Concrat benchmarks where the verifiers could produce meaningful output (e.g., atomics are a root cause for 4 of the benchmarks in Section 4.2 and occur in 5 benchmarks in Table 5). The characteristics of the entire Concrat suite are similar to those of the Subset, with the largest difference for property 3 of 21%. Moreover, each idiom is also present in at least one of the excluded benchmarks, validating that these idioms are prevalent across the entire Concrat benchmark suite.

Finding 3. We find that per-thread-data idioms and thread-joining schemes occur frequently in real-world programs. Features indicating the use of constructs that cause verifiers to fail are under-represented in SV-COMP tasks.

6 Discussion

Our study was conducted using data from SV-COMP 2023, with the extracted examples submitted as new benchmarks for the SV-COMP 2024 benchmark set. For fairness, we separately report both results from SV-COMP 2023, before our study, as well as for SV-COMP 2024. Since SV-COMP 2024 has now concluded, we can report its results, showing that the challenges that we found remain yet unsolved. Additionally, we elaborate on how the submission of these new benchmarks has already positively influenced the direction of the research community.

6.1 Progress since SV-COMP 2023

Tools. Compared to SV-COMP 2023, there were an additional two automated verifiers in the NoDataRace category in SV-COMP 2024. The results of the tools in SV-COMP 2024 are in Table 6. ESBMC is an SMT-based context-bounded model checker that we have included in the evaluation of 2024 results. However, PeSCo is based on CPAchecker with the addition of using machine learning for selecting algorithms. Thus, similarly to the Ultimate family of analyzers, we only evaluate the best-performing tool from the CPAchecker family.

Table 6. Positively scoring tools from SV-COMP 2024 and their results in the NoDataRace category. Correctly and incorrectly solved verification task counts are given by expected verdict (True or False).

			Correct		Incorrect		
Verification tool	Ref.	Approach	True	False	True	False	Score
Goblint	[69, 88]	Abstract interpretation	669	0	0	0	1,338
Dartagnan	[32, 60]	Bounded model checking (BMC)	495	154	1	0	1,112
UAutomizer	[38, 39]		557	106	0	0	1,220
UGemCutter	[28, 48]	Automata based model checking	536	144	0	1	1,200
UTaipan	[24, 34]		301	98	0	0	700
Theta	[1, 84]	Abstraction-based model checking	310	44	0	0	664
CPAchecker	[11, 21]	SMT-based model checking	249	56	0	0	554
PeSCo		Based on CPAchecker and machine learning	248	56	0	0	552
Locksmith	[62]	Constraint-based data-flow analysis	129	0	3	0	162
ESBMC	[29]	SMT-based Context-Bounded Model Checker	488	437	23	3	141
Deagle	[36, 37]	Satisfiability modulo theory (SMT)					Disqualified

Table 7. Experimental results of the selected tools' SV-COMP 2024 versions on the extracted examples.

Extracted example	Fig.	Goblint	Deagle	Dartagnan	UAutomizer	CPAchecker	Locksmith	Theta	ESBMC
per-thread-struct	1a	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
per-thread-array-ptr	1b	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
per-thread-array-index	1c	Inconcl	Inconcl	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
per-thread-array-init	1d	Inconcl	Correct	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
per-thread-struct-in-array	-	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
per-thread-index-inc	1e	Inconcl	Correct	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
per-thread-index-bitmask	-	Inconcl	Inconcl	LibFun	OOM	Unsupp	Inconcl	Parsing	Timeout
thread-local-value	2a	Correct	Correct	Unsupp	Unsupp	Unsupp	Inconcl	Parsing	Timeout
thread-local-value-cond*	-	Inconcl	Correct	Unsupp	Unsupp	Unsupp	Inconcl	Parsing	Timeout
thread-local-value-dynamic*	-	Inconcl	Correct	Unsupp	Unsupp	Unsupp	Inconcl	Parsing	Timeout
thread-local-pthread-value*	2b	Inconcl	Correct	Crash	LibFun	Unsupp	Correct	Parsing	Crash
thread-local-pthread-value-cond*	-	Inconcl	Correct	Crash	LibFun	Unsupp	Inconcl	Parsing	Crash
thread-join-array-dynamic	3a	Inconcl	Unsound	Unsupp	Timeout	Unsupp	Inconcl	Parsing	Timeout
thread-join-array-const	-	Inconcl	Unsound	Correct	Correct	Unsupp	Inconcl	Parsing	Unsound
thread-join-binomial	-	Inconcl	Unsound	Unsupp	OOM	Unsupp	Inconcl	Parsing	Timeout
thread-join-counter-outer	3b	Inconcl	Unsound	LibFun	OOM	Unsupp	Inconcl	Parsing	Timeout
thread-join-counter-inner	-	Inconcl	Unsound	LibFun	Timeout	Unsupp	Inconcl	Parsing	Timeout
atomic-gcc*	4a	Inconcl	Correct	Unsupp	Timeout	Unsupp	Correct	Parsing	Timeout
semaphore-posix	4b	Inconcl	Unsound	LibFun	LibFun	Unsupp	Unsound	Parsing	Timeout
value-barrier	4c	Inconcl	Correct	Unsupp	OOM	Unsupp	Inconcl	Parsing	Timeout

Correct — Soundly verified
Inconcl — Inconclusive
Unsound — Unsoundly verified
OOM — Out of memory
Timeout — Not finished within time limit
LibFun — Unsupported library function
Unsupp — Unsupported syntax
Parsing — Parsing failed
Crash — Tool crashed
* Extracted example does not have racy variations.

Results on the Micro-Benchmarks. As mentioned before, our extracted micro-benchmarks were accepted for SV-COMP 2024. The SV-COMP process is divided into three stages: the *benchmark submission phase*, the *training phase*, and the *evaluation phase* [8]. The benchmark set is finalized approximately 2 weeks before the competition (November 13th, 2023 for SV-COMP 2024). Following this, the tools undergo preliminary test runs, the *training phase*, to identify any major configuration flaws or implementation bugs prior to the main evaluation (November 23rd, 2023 for SV-COMP 2024). Authors of the tools thus have around 2 weeks to test their tools on the finalized benchmark set and address any identified issues. While authors may address these issues, the development of novel methods targeting the new benchmarks is not to be expected during such a limited time frame.

There has been no significant progress regarding our micro-benchmarks in the tools competing at SV-COMP 2024 [9], a fact evident from Table 7. Considering the results and the procedure outlined above, there has been no significant development aimed at addressing these challenges yet. Nonetheless, some progress has been made in enhancing the competing tools:

- Goblint is now sound on all of the micro-benchmarks.
- Deagle can verify more of the challenges than last year but is unfortunately also more unsound due to that.
- Dartagnan can now verify the thread-join-array-const benchmark.
- UAutomizer does not handle thread local values, and thus mistakenly marked some thread-local benchmarks with True verdicts as False in 2023. They resolved this issue by indicating that they do not yet support thread-local variables, providing an Unknown verdict instead.

Table 8. Experimental results of the selected tools with SV-COMP 2024 settings on the included Subset of the Concrat benchmark set with 2 hours of CPU time, 24 GB of RAM, and 8 CPU cores. The second column indicates program size in lines of code (LoC).

Benchmark	LoC	Goblint	Deagle	Dartagnan	UAutomizer	CPAchecker	Locksmith	Theta	ESBMC
axel*	5,848	Timeout	True	Unsupp	Unsupp	Unsupp	Crash	Parsing	Crash
C-Thread-Pool*	710	Unknown	True	Unsupp	Unsupp	Unsupp	Unknown	Parsing	Timeout
dnspod-sr	9,259	Timeout	Unknown	LibFun	Unsupp	Unsupp	Unknown	Parsing	True
EasyLogger*	2,011	Unknown	OOM	LibFun	LibFun	Unsupp	Timeout	Parsing	OOM
fzy	2,621	Unknown	Timeout	Unsupp	Parsing	Unsupp	Unknown	Parsing	Crash
klib*	716	Unknown	Timeout	LibFun	Unsupp	Unsupp	Unknown	Parsing	False
level-ip	5,414	Unknown	OOM	Unsupp	LibFun	Unsupp	Crash	Parsing	Crash
libfaketime	521	True	True	LibFun	Crash	Unsupp	True	Parsing	True
lmdb	10,827	Crash	True	LibFun	LibFun	Unsupp	Timeout	Parsing	Crash
Mirai-Source-Code	1,839	Crash	LibFun	Unsupp	Parsing	Unsupp	Unknown	Parsing	Unsupp
nnn	12,091	Unknown	OOM	LibFun	LibFun	Timeout	Unknown	Parsing	Crash
phpspy	19,390	Crash	Crash	Unsupp	Parsing	Unsupp	Timeout	Parsing	Crash
pigz	9,118	OOM	Unknown	Parsing	Unsupp	Timeout	Timeout	Parsing	Crash
ProcDump-for-Linux*	4,152	Unknown	Timeout	Parsing	LibFun	Unsupp	Unknown	Parsing	OOM
Remotery*	7,212	Crash	Timeout	Unsupp	LibFun	Unsupp	Timeout	Parsing	Crash
streem*	20,169	Crash	Timeout	Crash	LibFun	Unsupp	Timeout	Parsing	Crash
the_silver_searcher*	7,242	Crash	Crash	Parsing	Parsing	Unsupp	Unknown	Parsing	Crash
uthash	817	True	Unknown	LibFun	LibFun	Unsupp	Unknown	Parsing	OOM

True — Tool claims data race freedom
False — Tool claims existence of data race
Unknown — Neither claimed verified nor refuted
OOM — Out of memory
Timeout — Not finished within time limit

LibFun — Unsupported library function
Unsupp — Unsupported syntax
Parsing — Parsing failed
Crash — Tool crashed

* ThreadSanitizer found a data race.

Deagle was disqualified from SV-COMP 2024 due to the forbidden use of identifiers contained within the verification tasks for identifying groups of tasks for setting the parameters of the verification engine.

Results on Concrat. We also evaluated the tools with SV-COMP 2024 settings on the Concrat benchmark set, resulting in Table 8. In comparison to the evaluation of tools from SV-COMP 2023 (as shown in Table 2), Deagle now claims data race freedom in 4 programs, whereas previously it claimed it in none. However, caution is warranted as Deagle exhibits significant unsoundness on the extracted micro-benchmarks, indicating a potential for falsely asserting data race freedom in larger programs. The new tool, ESBMC, claimed data race freedom in two of the Concrat benchmarks. Furthermore, unlike any other tools under evaluation, ESBMC also claims the existence of a data race in one program. However, the error trace it prints is invalid: the data race occurs at the beginning of the program before any threads are created.

6.2 Towards a World Free from Data Races: What is the Path Forward?

To realize the vision of a world free from data races, we need to develop abstractions and algorithms that can handle idioms derived from real-world programs, including those identified in this work. We raise the question of how to evaluate progress towards the analysis of real-world programs, emphasizing the need for both micro- and macro-benchmarks. Finally, we acknowledge the possibility that we may ultimately need to move to safer languages, or at least borrow some of their features.

Developing Abstractions and Algorithms for Relevant Idioms. We have identified the core semantic properties of our extracted idioms to clarify the verification conditions that need solving to handle them. In many cases, the theoretical underpinnings already exist—for example, concurrent separation logic [15]. However, challenges arise when the separation depends on dynamic parameters passed to `pthread_create`. Still, relating addresses with integer arguments is solvable [35, 75, 77]. Similarly, existing array-segment analyses [20] can handle the simpler patterns involved in thread joining. It is not trivial to combine these techniques and apply them to race freedom verification, but our hope is that the challenge is manageable if the community knows what to focus on.

The SV-COMP benchmark suite includes many regression suites from tool developers, which reflects the idioms that these tools are designed to handle. Our findings reveal that certain features of real-world programs are significantly underrepresented in the SV-COMP benchmark suite. In particular, nearly half of the real-world programs rely on per-thread heap separation; however, we found only six tasks in the SV-COMP that involve data flow from dynamically allocated structures to created threads. We consider this the most significant gap in current tool capabilities.

Evaluation on Micro- and Macro-Benchmarks. To move beyond verifying only the challenging idioms, we must also ensure that the new techniques scale to real-world programs. Thus, it is important to evaluate tools on both micro- and macro-benchmarks. A rich suite of micro-benchmarks, such as that of SV-COMP, is particularly important for detecting unsoundness in the tools. On the other hand, the real-world benchmarks require scalable and composable solutions. Composability is particularly important because we have expressed some idioms in terms of their core components, but as we point out, the idioms are often intertwined in real-world programs. For example, a thread pool implementation in our micro-benchmark contains a single array of thread-local data, but in the real-world program, there may be multiple thread pools, each with a dynamically allocated struct containing multiple arrays per thread pool. Thus, it is essential to integrate novel algorithms and abstractions in a way that handles such compositions.

There is a trade-off between the two types of benchmarks, and it is important to evaluate tools on both types of benchmarks to ensure that they are soundly handling the core idioms and that the methods scale to real-world programs. While verification contests, such as SV-COMP, are important for evaluating tools on micro-benchmarks, the community currently lacks a process to similarly evaluate progress on macro-benchmarks.

Language-Support for Verification. One potential conclusion of our work is that fully automated race freedom verification for C programs is indeed a lost cause, and we should advocate for the use of safer languages. Rust's "fearless concurrency" [47] addresses disjointness for freshly allocated structs by enforcing memory ownership rules. Short of translating all C programs to Rust, one may consider adopting ownership types for verifying C programs [71]. For thread joining, Java-style foreach loops are easier to analyze for completeness of iteration. C's new `counted_by` attribute, or an analyzer-specific annotation, could be used to indicate live thread counters. Yet, our idioms include some that even Rust does not guarantee—ownership in Rust operates at the level of the entire array, and does not ensure essential properties where threads own disjoint parts of an array.

6.3 Threats to Validity

The conclusions of this particular work are about state-of-the-art static automated race freedom verification tools. Our principal claim is that our twelve challenging idioms encapsulate fundamental patterns that these verifiers cannot verify. The smaller, easily benchmarkable instances of constructs that we propose can inspire research that contributes to needed advances in knowledge by allowing researchers to develop analyses tailored to them. We have shown that current state-of-the-art tools do struggle with these instances and that they do occur in real-world programs.

Internal Validity. The execution of the tools and interpretation of their verdicts was handled by BenchExec [12]. We used the artifact packages from the official SV-COMP archive. The identification of the idioms relies on expert knowledge and is difficult to replicate. We provided a quantitative evaluation in Section 5.2 to mitigate this risk.

The state-of-the-art verifiers that participate in SV-COMP need to adhere to the C99 standard [44], which lacks a formal memory model. Current SV-COMP tasks can, however, be verified using sequential consistency, because these tasks do not have C11 atomics, so the DRF guarantee can be assumed. Even though tools like Dartagnan and Deagle are capable of handling weak memory models [32, 36], these capabilities are thus currently not evaluated by SV-COMP [59]. While addressing weak memory is an important challenge, its existence does not invalidate our results for the existing large collection of programs in the world that do not rely on relaxed semantics.

External Validity. In Section 4.2 we selected a benchmark set from the literature that has been constructed by searching GitHub repositories under certain criteria. While these criteria aim to cover a wide range of programs found in the wild, it is possible that the benchmark set is not fully representative of all real-world programs. One might consider this to be a threat to external validity—our results might not generalize to every domain or type of software. However, this threat does not invalidate the fact that we have identified idioms that indeed occur in some real-world programs. Importantly, our selected benchmark set falls into one specific class of software: primarily general-purpose, open-source programs, which represent a significant and widely used domain in the software ecosystem. Similar studies could be conducted in other domains, such as embedded operating systems or specialized libraries, to uncover idioms and synchronization patterns unique to those contexts. Despite this focus, our set of idioms provides a solid foundation and a reasonable starting point for the development of novel techniques.

We were also compelled to eliminate some benchmarks from the set because they were incomplete or single-threaded. The quantitative evaluation indicates that the relevant features are present in the eliminated benchmarks as well. However, to further mitigate the risk of external validity, we have made our benchmark set open, with an opportunity and encouragement for others to contribute more real-world programs to the benchmark set in the future.

Our choice of tool to aid the extraction process was Goblint. One may question if choosing a different tool would have led to different results. For the sake of argument, we fix the benchmark set—the representativeness of the chosen benchmarks among the entire class of programs of interest is an orthogonal issue. There is a universe of underlying idioms distributed across the benchmark set. The main result of this work is the idioms from the benchmarks that we have identified.

Let us turn our attention to idioms in the universe that are somewhere in the benchmark set but which we do not flag in our Goblint-centered idiom extraction process. There are two possibilities: Goblint already handles them, or Goblint does not handle them. In the first case, such idioms may indeed be important, but, by definition, are not beyond the state-of-the-art. (Most of Goblint's capabilities are reflected in its comprehensive regression suite, which we have previously integrated into SV-COMP.) The second case is admittedly possible, but only for benchmarks where we have, in the present work, extracted idioms that Goblint cannot handle. If we assume that Goblint is sound, then using a different tool than Goblint for idiom extraction could, at worst, extract a *different* set of idioms from benchmarks where we have identified at least one idiom using Goblint. (An unsoundness in Goblint could result in our work overlooking idioms from one of the two benchmarks on which Goblint has returned True).

Thus, after handling the idioms we have identified, it is necessary to revisit the original benchmarks when these idioms have been solved, to identify idioms that may have been obscured by ours.

7 Related Work

In this section, we expand on related work. In particular, we present additional data race detection tools and benchmark suites.

Evaluating Data Race Detection Tools. Yu et al. [92] conducted a comparative evaluation of ten data race detection tools on small to medium-sized C programs. While both our work and theirs evaluated tools on real-world programs, their evaluation considered dynamic data race detectors, while ours considers static data race freedom verifiers. Another work that also considers real applications is by Schimmel et al. [72], using 25 repositories that include data races to evaluate their selected tools. However, they consider Java repositories and dynamic race detectors for Java. (Interestingly, even using dynamic tools, they reported many false positives.)

The SAMATE group investigated the effectiveness of static analysis bug-finding tools on real-world software, presenting the results at a series of workshops [70] from 2008 through 2019. We also statically analyze real-world software but aim to prove freedom from data races. There have been a number of empirical investigations on why and how static analysis tools are applied in practice [6, 19, 46]. These studies highlight many usability challenges to integrating these tools into the development process, but they also confirm that lack of precision is a major issue preventing wider application of static analysis tools.

Analyzing False Positives for C. Recall that our goal is to understand conceptual barriers to static verification of real-world programs. The authors of the Locksmith static race detector for C [61] reported that false positives occurred mainly due to two categories of coding idioms. The first category concerns idioms like thread joining or signaling, where a parent thread accesses previously shared data after its child threads have terminated. The second involves idioms where thread-local data is held in a global data structure that is indexed by thread identifiers. In follow-up work [62], they also identify semaphores and inline atomic assembly instructions as causes of false positives. They state that LP-Race [82], a static analysis tool that reduces the problem of race detection to linear programming, does not report some of these false positives. LP-Race can handle semaphores and thread joining. However, due to the reduction to linear programming, it still reports false positives for loops that create an unbounded number of threads. In addition, it also reports false positives on thread-local data. Finally, LP-Race cannot verify the initialization of thread-local data that is later shared.

Voung et al. [90] thoroughly analyze a randomly selected subset of warnings from the Relay race detector, uncovering that most of these warnings were, in fact, false positives. Additionally, they categorize these false positives based on the coding idioms used to prevent race conditions. The six reported categories were initialization, unlikely aliasing, unsharing, re-entrant locks, non-parallel threads and conditional locking. They emphasize the need for advanced analyses that are concurrency-, path- and shape-sensitive, as well as the significance of jointly considering multiple aspects when verifying a coding idiom. The Goblint analyzer targeted some of these limitations [87], such as conditional locking [89] and spurious aliasing [76], but their evaluation reports false positives due to unsupported protection mechanisms, environmental assumptions, and lack of precision [88]. Although the causes of false positives are identified in these evaluations, we aim to collect specific patterns of thread joining and task decomposition that are often used.

More generally, the process of identifying the cause of false positives and extracting regression tests is a common process, though it is often manual and ad-hoc. There have been efforts to automate and provide tool support for this process [41, 55, 67]. To produce concise regression cases, automated program minimization tools can be helpful [66, 81], though manual effort is needed to obtain human-readable examples. Finally, to generate variations of a given kernel, mutation

tools [93] can be applied, but we are mainly interested in mutations that expose unsound reasoning, which requires manual intervention. More research in these areas, with a focus on extracting human-readable test cases, would be beneficial to support the development of static analysis tools.

Concurrency Bugs in Real-World Programs. Others have investigated concurrency bugs in real-world programs. In the Go context, Tu et al. [85] conducted a systematic study on concurrency bugs, while Chabbi and Ramanathan [17] focused on data race bugs and their patterns. Qin et al. [63] studied concurrency issues for real-world Rust programs. However, both Go and Rust are opinionated about concurrency patterns—Go encourages communication via channels instead of shared memory, while Rust's type system statically ensures race freedom in safe code. Our research instead focuses on the C language, specifically exploring the data-race-free constructs of the language as found in real-world programs. C has completely different affordances for concurrency; it was not designed to help programmers avoid races.

Benchmarking Concurrency Bugs. The awareness of the importance of benchmarking and replication in programming language research is increasing, so there have been many other efforts to address these challenges. The JaConTeBe suite, by Lin et al. [52], collects concurrency bugs from open-source Java projects while following the five guidelines by Lu et al. [53] for preparing benchmark suites for bug detection tools. Gao et al. [31] point out that although JaConTeBe contains 47 concurrency bugs, only 19 are data race bugs. They propose a new benchmark set, JBench, tailored for evaluating Java data race detection tools. Similar to our approach, JBench combines Java programs with data race bugs from the literature as well as real-world applications. However, we started from the SV-COMP competition suite as well as the Concrat real-world suite; both of these suites were curated by others. SV-COMP collects community submissions while Concrat aims to exhaustively include all sufficiently-starred programs up to a certain size. Also, our key contribution of a set of challenging problems—informed by the design of race detection tools—is absent from their work.

A key difference between our work and these studies is that we do not propose a benchmark set for evaluating tools that detect data race bugs, but instead for evaluating tools that verify *data race freedom*. Our proposed benchmark set consists of real-world C programs, as well as kernel programs with examples of unsolved verification problems derived from real-world C programs. The programs are not proposed as a static benchmark set but collected into a living continuous benchmarking process, a practical analog to a living systematic review.

Pointer Analyses for (Lock-Based) Race Freedom Verification. What makes race verification in C particularly challenging is the use of pointers. Determining whether two expressions in a program may access the same memory location requires alias analysis, a well-known and longstanding challenge in static analysis [40]. Similarly, for time-separated accesses, establishing whether synchronization exists between two accesses is also a well-studied problem [2, 23, 94]. Both alias analysis and synchronization analysis are undecidable in general [64, 65], and the interplay between the two dimensions of separation—in space and time—is particularly challenging, as we observed in this paper.

We presented mainly idioms that did not rely on locking because a great amount of research has focused on handling various locking idioms. However, even establishing that a pair of accesses is safe due to the correct use of locking, while allowing both access expressions and locking expressions to involve pointers, is already a considerable challenge. An analysis needs to show that whenever two access expressions *may* alias, the addresses of the acquired locks *must* be equal. While there are many excellent techniques for scalable may-alias analysis, and similarly so for must-equality analysis of addresses, the combination of interprocedural may-alias and must-equality analyses is

still a challenge for scalable static data race analysis. Bodden et al. [13] outline an *intra*procedural approach, but combining these analyses in the interprocedural context is particularly important—in some programs, locks are acquired by one function and released by another. Specific solutions for the context-sensitive computation of joint may-alias and must-equality information have been proposed, such as the relative lockset analysis of Relay [90], the polymorphic type system of Locksmith [61], the side-effecting constraint system of Goblint [4], or even the conditional must-not-alias analysis of the Chord analyzer for Java [56]. The last paper provided a conditional perspective that could generalize well to the time-based idioms.

Dynamic and unsound tools. Dynamic race detection tools like ThreadSanitizer [78, 79] are generally unsound but can provide concrete evidence of a data race. Practical static analyzers, including Coverity Scan and the Java-based tool RacerDX, abandon soundness in favor of producing high-signal data race reports that developers can act upon [33]. As the tools in our experiment failed to pinpoint any actual races in the real-world programs, developing rigorous formal approaches to bug detection is also a welcome development [49, 58]. The frequent appearance and subsequent patching of race-related vulnerabilities in released software highlight the limitations of current approaches, both static and dynamic. This suggests that while unsound methods have their strengths, they may not yet be sufficient on their own, in practice, to prevent data race bugs post-release. Our work focuses on sound static data race verification methods, aiming to complement dynamic and unsound approaches for better prevention of data races in real-world programs.

8 Conclusion

To identify fundamental challenges for today's static data race freedom verifiers, we considered a set of 18 real-world programs drawn from the Concrat benchmark suite. We present a principled set of 20 coding idioms, extracted from these programs, and evaluated them on 7 state-of-the-art verifiers from SV-COMP 2023. The verifiers are specialized for competition problems and generally could not handle real-world programs—2 verifiers successfully verified race freedom in a total of 3 programs, with resource exhaustion and superficial issues causing failures. The verifiers did better on the extracted idioms than on the complete programs, with some verifier able to handle 8 of the 20 idioms; the remaining 12 idioms are beyond today's state-of-the-art.

> **Takeaway message.** Our learning from this work is that, while lock-based idioms for avoiding data races are well-supported by existing data race verification tools, real-world multithreaded C applications extensively use other idioms as well. Specifically, idioms involving per-thread structs and arrays, where each thread has exclusive access to a segment of the array, are commonly used in practice and are not well-supported by current tools. It is necessary (though admittedly perhaps not sufficient) to support these idioms if one is to verify race freedom of C programs.

This work takes an important first step—that of problem identification—towards the vision of static data race verification for all real-world C programs. We call on the community to design and implement techniques to handle the idioms identified in this work, which are a necessary prerequisite for achieving this vision.

Data-Availability Statement

The benchmark sets, tools under evaluation, scripts for reproduction and instructions of use that support the data in Tables 2, 4, 5, 10 and 14 are available on Zenodo [42].

Acknowledgments

The authors thank the anonymous referees for their valuable comments and suggestions. We also gratefully acknowledge SIGPLAN-M for facilitating the introduction between the authors. This work was supported by the European Union and the Estonian Research Council via projects PRG2764 and TEM-TA119.

References

[1] Zsófia Ádám, Levente Bajczi, Mihály Dobos-Kovács, Ákos Hajdu, and Vince Molnár. 2022. Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 474–478. doi:10.1007/978-3-030-99527-0_34

[2] Elvira Albert, Samir Genaim, and Pablo Gordillo. 2015. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–89. doi:10.1007/978-3-662-48288-9_5

[3] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. 2011. Making Software Verification Tools Really Work. In *Automated Technology for Verification and Analysis (Lecture Notes in Computer Science)*, Tevfik Bultan and Pao-Ann Hsiung (Eds.). Springer, Berlin, Heidelberg, 28–42. doi:10.1007/978-3-642-24372-1_3

[4] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2012. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–172.

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/1926385.1926394

[6] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 470–481. doi:10.1109/SANER.2016.105

[7] Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 375–402. doi:doi.org/10.1007/978-3-030-99527-0_20

[8] Dirk Beyer. 2023. Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 495–522. doi:10.1007/978-3-031-30820-8_29

[9] Dirk Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 299–329. doi:10.1007/978-3-031-57256-2_15

[10] Dirk Beyer and Karlheinz Friedberger. 2016. A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker. *Electronic Proceedings in Theoretical Computer Science* 233 (12 2016). doi:10.4204/EPTCS.233.6

[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–190. doi:10.1007/978-3-642-22110-1_16

[12] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2017. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21, 1 (nov 2017), 1–29. doi:10.1007/s10009-017-0469-y

[13] Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Object Representatives: A Uniform Abstraction for Pointer Information. In *Proceedings of the 2008 International Conference on Visions of Computer Science: BCS International Academic Conference* (London, UK) *(VoCS'08)*. BCS Learning & Development Ltd., Swindon, GBR, 391–405. doi:10.14236/ewic/VOCS2008.32

[14] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 68–78. doi:10.1145/1375581.1375591

[15] Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (Aug. 2016), 47–65. doi:10.1145/2984450.2984457

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224. http://usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[17] 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 474–489. doi:10.1145/3519939.3523720

[18] Cheng Chen, Shuo Li, and Zhijun Ding. 2022. Automatic Modeling Method for PThread Programs Based on Program Dependence Net. In *Proceedings of the 2021 3rd International Conference on Software Engineering and Development* (Xiamen, China) *(ICSED '21)*. Association for Computing Machinery, New York, NY, USA, 9–18. doi:10.1145/3507473.3507475

[19] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 332–343. doi:10.1145/2970276.2970347

[20] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 105–118. doi:10.1145/1926385.1926399

[21] Matthias Dangl, Stefan Löwe, and Philipp Wendler. 2015. CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 423–425. doi:10.1007/978-3-662-46681-0_34

[22] David Delmas and Jean Souyris. 2007. Astrée: From Research to Industry. In *Static Analysis*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 437–451. doi:10.1007/978-3-540-74061-2_27

[23] Peng Di, Yulei Sui, Ding Ye, and Jingling Xue. 2015. Region-Based May-Happen-in-Parallel Analysis for C Programs. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP) (ICPP '15)*. IEEE Computer Society, USA, 889–898. doi:10.1109/ICPP.2015.98

[24] Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Frank Schüssele, and Andreas Podelski. 2023. Ultimate Taipan and Race Detection in Ultimate. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 582–587. doi:10.1007/978-3-031-30820-8_40

[25] 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

[26] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 57–72. doi:10.1145/502034.502041

[27] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound Sequentialization for Concurrent Program Verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 506–521. doi:10.1145/3519939.3523727

[28] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound Sequentialization for Concurrent Program Verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 506–521. doi:10.1145/3519939.3523727

[29] Mikhail R. Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. 2019. ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 209–213. doi:10.1007/978-3-030-17502-3_15

[30] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 888–891. doi:10.1145/3238147.3240481

[31] Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. 2018. Jbench: A Dataset of Data Races for Concurrency Testing. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 6–9. doi:10.1145/3196398.3196451

[32] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 355–365. doi:10.1007/978-3-030-25540-4_19

[33] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A true positives theorem for a static race detector. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 57:1–57:29. doi:10.1145/3290370

[34] Marius Greitschus, Daniel Dietsch, and Andreas Podelski. 2017. Loop Invariants from Counterexamples. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 128–147. doi:10.1007/978-3-319-66706-5_7

[35] Sumit Gulwani and Ashish Tiwari. 2006. Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (LNCS, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 279–293. doi:10.1007/11693024_19

[36] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo Ordering Consistency Theory for Multi-Threaded Program Verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1264–1279. doi:10.1145/3453483.3454108

[37] Fei He, Zhihang Sun, and Hongyu Fan. 2022. Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 424–428. doi:10.1007/978-3-030-99527-0_25

[38] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. 2023. Ultimate Automizer and the CommuHash Normal Form. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 577–581. doi:10.1007/978-3-031-30820-8_39

[39] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–52. doi:10.1007/978-3-642-39799-8_2

[40] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE '01)* (Snowbird, Utah, USA). Association for Computing Machinery, New York, NY, USA, 54–61. doi:10.1145/379605.379665

[41] Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, Simmo Saan, and Vesal Vojdani. 2024. Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pasadena, CA, USA) *(Onward! '24)*. Association for Computing Machinery, New York, NY, USA, 130–146. doi:10.1145/3689492.3690053

[42] Karoliine Holter, Simmo Saan, Patrick Lam, and Vesal Vojdani. 2024. *Sound Static Data Race Verification for C: Is the Race Lost?* doi:10.5281/zenodo.10903393 Artifact.

[43] Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Computer Society, Los Alamitos, CA, USA, 716–728. doi:10.1109/ICSE48619.2023.00069

[44] ISO/IEC. 1999. Programming Languages – C. ISO/IEC 9899:1999. https://www.iso.org/standard/29237.html International Standard.

[45] ISO/IEC. 2011. ISO International Standard ISO/IEC 9899:2011.

[46] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

[47] Steve Klabnik and Carol Nichols. 2024. *The Rust Programming Language*. No Starch Press.

[48] Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. 2022. Ultimate GemCutter and the Axes of Generalization. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 479–483. doi:10.1007/978-3-030-99527-0_35

[49] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (apr 2022), 27 pages. doi:10.1145/3527325

[50] 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

[51] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 443–457. doi:10.1145/3009837.3009857

[52] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 178–189. doi:10.1109/ASE.2015.87

[53] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5. Chicago, Illinois. https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/63-lu.pdf

[54] Antoine Miné. 2012. Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs. *Log. Methods Comput. Sci.* 8, 1 (2012). doi:10.2168/LMCS-8(1:26)2012

[55] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2024. Easing Maintenance of Academic Static Analyzers. arXiv:2407.12499 [cs.PL] https://arxiv.org/abs/2407.12499

[56] Mayur Naik and Alex Aiken. 2007. Conditional Must Not Aliasing for Static Race Detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 327–338. doi:10.1145/1190216.1190265

[57] Mattias Nyberg, Dilian Gurov, Christian Lidström, Andreas Rasmusson, and Jonas Westman. 2018. Formal Verification in Automotive Industry: Enablers and Obstacles. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, 139–158. doi:10.1007/978-3-030-03427-6_14

[58] Peter W. O'Hearn. 2019. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. doi:10.1145/3371078

[59] Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II* (Dublin, Ireland). Springer-Verlag, Berlin, Heidelberg, 378–382. doi:10.1007/978-3-030-45237-7_24

[60] Hernán Ponce-de León, Thomas Haas, and Roland Meyer. 2021. Dartagnan: Leveraging Compiler Optimizations and the Price of Precision (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 428–432. doi:10.1007/978-3-030-72013-1_26

[61] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. 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, 320–331. doi:10.1145/1133981.1134019

[62] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (jan 2011), 55 pages. doi:10.1145/1889997.1890000

[63] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. doi:10.1145/3385412.3386036

[64] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (sep 1994), 1467–1471. doi:10.1145/186025.186041

[65] G. Ramalingam. 2000. Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (mar 2000), 416–430. doi:10.1145/349214.349241

[66] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. doi:10.1145/2254064.2254104

[67] Xavier Rival. 2005. Understanding the Origin of Alarms in Astrée. In *Static Analysis*, Chris Hankin and Igor Siveroni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 303–319.

[68] Eric F. Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. 2016. On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 132–143. doi:10.1145/2884781.2884835

[69] Simmo Saan, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. 2023. Goblint: Autotuning Thread-Modular Abstract Interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 547–552. doi:10.1007/978-3-031-30820-8_34

[70] SAMATE. 2021. Static Analysis Tool Exposition (SATE). https://www.nist.gov/itl/ssd/software-quality-group/samate/static-analysis-tool-exposition-sate. Accessed 16 November 2023.

[71] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd*

ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. doi:10.1145/3453483.3454036

[72] Jochen Schimmel, Korbinian Molitorisz, and Walter F. Tichy. 2013. An Evaluation of Data Race Detectors Using Bug Repositories, In 2013 Federated Conference on Computer Science and Information Systems. *2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013*, 1361–1364. https://ieeexplore.ieee.org/document/6644193

[73] Michael Schwarz and Julian Erhard. 2024. The digest framework: concurrency-sensitivity for abstract interpretation. *International Journal on Software Tools for Technology Transfer* (28 Dec 2024). doi:10.1007/s10009-024-00773-y

[74] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. 2021. Improving Thread-Modular Abstract Interpretation. In *Static Analysis*, Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi (Eds.). Springer International Publishing, Cham, 359–383.

[75] Helmut Seidl, Julian Erhard, Michael Schwarz, and Sarah Tilscher. 2024. *2-Pointer Logic*. Springer Nature Switzerland, Cham, 281–307. doi:10.1007/978-3-031-56222-8_16

[76] Helmut Seidl and Vesal Vojdani. 2009. Region Analysis for Race Detection. In *Proceedings of the 16th International Symposium on Static Analysis, SAS '09* (Los Angeles, CA) *(LNCS, Vol. 5673)*. Springer, 171–187. doi:10.1007/978-3-642-03237-0_13

[77] Helmut Seidl, Vesal Vojdani, and Varmo Vene. 2009. A Smooth Combination of Linear and Herbrand Equalities for Polynomial Time Must-Alias Analysis. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (LNCS, Vol. 5850)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 644–659. doi:10.1007/978-3-642-05089-3_41

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

[79] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic Race Detection with LLVM Compiler. In *Runtime Verification*, Sarfraz Khurshid and Koushik Sen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 110–114. doi:10.1007/978-3-642-29860-8_9

[80] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 9–19. doi:10.1109/ICSE.2015.24

[81] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. doi:10.1145/3180155.3180236

[82] Tachio Terauchi. 2008. Checking race freedom via linear programming. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/1375581.1375583

[83] The Open Group. 1997. *pthread_cond_wait*. The Open Group. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_wait.html IEEE Std 1003.1, 1996 Edition.

[84] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: A Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (Vienna, Austria) *(FMCAD '17)*. FMCAD Inc, Austin, Texas, 176–179. doi:10.23919/FMCAD.2017.8102257

[85] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878. doi:10.1145/3297858.3304069

[86] Edwin Török. 2023. Targeted Static Analysis for OCaml C Stubs: eliminating gremlins from the code. doi:10.48550/arXiv.2307.14909 arXiv:2307.14909 [cs].

[87] Vesal Vojdani. 2010. *Static Data Race Analysis of Heap-Manipulating C Programs*. Ph. D. Dissertation. University of Tartu. http://hdl.handle.net/10062/15866

[88] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static Race Detection for Device Drivers: The Goblint Approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 391–402. doi:10.1145/2970276.2970337

[89] Vesal Vojdani and Varmo Vene. 2009. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.* 30 (2009), 141–155. http://ac.inf.elte.hu/Vol_030_2009/141.pdf

[90] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 205–214. doi:10.1145/1287624.1287654

[91] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, USA, 590–604. doi:10.1109/SP.2014.44

[92] Zhen Yu, Zhen Yang, Xiaohong Su, and Peijun Ma. 2017. Evaluation and comparison of ten data race detection techniques. *International Journal of High Performance Computing and Networking* 10, 4-5 (2017), 279–288. doi:10.1504/IJHPCN.2017.086532

[93] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 237–249. doi:10.1145/3611643.3616272

[94] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-Happen-in-Parallel Analysis with Static Vector Clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 228–240. doi:10.1145/3168813

A Tools selected from SV-COMP 2023 and their engineering statistics

Verification tool	Start year[1]	Nr. of contributors[2]	Nr. of releases	Total commits[3]	Commits in 2023	Latest commit[3]
Goblint	2007	66	17	16,342	3,182	Aug 2024
Deagle	2021	2	4	5	1	Apr 2024
Dartagnan	2017	9	9	4,647	181	Aug 2024
UAutomizer UGemCutter UTaipan	2013	78	30	28,656	1,705	Aug 2024
CPAchecker	2009	126	22	34,844	1,658	Aug 2024
Locksmith	2006	4		27	0	Nov 2021
Theta	2016	25	107	5,617	183	Aug 2024

[1] Minimum year of first git commit and first paper.
[2] Maximum contributors of git repository and explicit authors list.
[3] As of Aug 14 2024.

B Fixes done in the Concrat benchmarks

We fixed syntactic issues in the Concrat benchmarks as follows:

(1) 14 programs contained large 64-bit unsigned integer constants without the `ULL` suffix, as required by C99 and beyond [45]. These caused parsing errors for standards-compliant C parsers. Therefore, we added the suffix where required.
(2) 3 programs contained re-declarations of compiler builtins, which are forbidden by LLVM and caused parsing errors for LLVM-based tools. These C declarations are non-defining, so we removed them without affecting the behavior of the programs.
(3) 2 programs contained multiple definitions of an `inline` function. These were identical duplicates. Therefore, we removed all but one definition.

We only fixed the syntactic issues and chose not to create mocks, as modeling of the environment can be a significant source of false positives, which would be a confounding factor in our study. Table 9 details the benchmarks to which these syntactic fixes were applied.

Table 9. List of benchmarks where the described syntactic fixes were applied.

Fix	Affected benchmarks
(1)	Cello, Chipmunk2D, Remotery, brubeck, kona, lmdb, minimap2, pg_repack, phpspy, pigz, streem, sysbench, wrk, zmap
(2)	Mirai-Source-Code, kona, streem,
(3)	sysbench, zmap

C Excluded Concrat benchmarks (28)

Benchmark	Reason for exclusion
AirConnect	Incomplete: missing `ixml*` functions
brubeck	Incomplete: missing `rd_kafka_*` and `json_*` functions
cava	Incomplete: missing `iniparser_*`, `fftw_*`, `pa_*` and `snd_*` functions
Cello	Incomplete: no `main` function
ChipMunk2D	Incomplete: no `main` function
clib	Incomplete: missing `curl_*` functions
dump1090	Incomplete: missing `rtlsdr_*` functions
kona	Single-threaded: no `pthread_create` call
libaco	Incomplete: uses external assembly code
libfreenect	Incomplete: missing `freenect_*`, `gl*` and `glut*` functions
libqrencode	Single-threaded: no `pthread_create` call
minimap2	Incomplete: missing `ksw_*` functions
neural-redis	Incomplete: no `main` function
pg_repack	Single-threaded: no `pthread_create` call
pianobar	Incomplete: missing curl, `json_*`, `ao_*`, `gcry_*` and `av_*` functions
pingfs	Incomplete: missing `fuse_*` and `gai_*` functions
proxychains	Incomplete: no `main` function
proxychains-ng	Incomplete: no `main` function
sc	Single-threaded: no `pthread_create` call
shairport	Incomplete: missing `snd_*`, `pa_*`, `ao_*`, `MD5_*` and `BIO_*` functions
siege	Incomplete: missing OpenSSL functions
snoopy	Incomplete: missing `snoopy_*` functions
sshfs	Incomplete: missing `fuse_*` and `g_*` functions
stud	Single-threaded: no `pthread_create` call
sysbench	Incomplete: missing `lua_*` functions
vanitygen	Incomplete: missing curl and OpenSSL functions
wrk	Incomplete: missing OpenSSL and lua functions
zmap	Incomplete: missing `log_*`, `pcap_*` and `json_*` functions

D Concrat benchmarks' descriptions

Axel A command-line program that accelerates HTTP/FTP downloads by using multiple connections for one file. It uses concurrency to download different parts of the file simultaneously through multiple threads.

C-Thread-Pool A library providing a thread pool implementation to execute tasks in parallel. It manages a finite number of threads that run tasks concurrently, improving performance for multithreaded applications.

Dnspod-sr A lightweight and fast DNS relay server. It handles multiple DNS queries concurrently to improve resolution speed and reduce response time.

EasyLogger A simple, efficient, and modular logging utility for C projects that ensures thread-safe logging operations across different threads in an application.

Fzy A fast, simple fuzzy text selector for the terminal with an advanced scoring algorithm. It performs multiple matching operations in parallel, increasing the search efficiency.

Klib A standalone and lightweight implementation of several generic data structures. The structures provided can be used in concurrent programming to manage shared data.

Level-ip A userspace level 3 network stack that processes multiple network packets simultaneously in a concurrent manner.

Libfaketime A tool for intercepting system calls to retrieve the current time, handling simultaneous time modification requests in multithreaded applications.

LMDB An ultra-fast, ultra-compact key-value embedded data store that supports multithreaded environments by allowing concurrent read access and serialized write access.

Mirai-Source-Code The source code for the Mirai botnet, is designed to manage communications and attacks across a large number of bots concurrently.

Nnn A fast and resource-sensitive file manager that uses asynchronous notifications and multithreaded file operations to enhance performance.

Phpspy A low-overhead sampling profiler for PHP that samples multiple threads of a PHP application concurrently.

Pigz A parallel implementation of gzip that divides the compression task into multiple threads to utilize multiple processors and cores.

ProcDump-for-Linux A tool for creating core dumps of applications based on performance triggers, monitoring multiple triggers, and managing dumps concurrently.

Remotery A real-time CPU/GPU profiler that collects performance data from different threads in real-time.

Streem A stream-based concurrent scripting language that processes data streams in parallel and constructs data pipelines easily.

The Silver Searcher A code-searching tool that uses multiple threads to search through large codebases quickly.

Uthash A hash table for C structures, often used in concurrent applications where synchronization mechanisms are applied externally.

Table 10. Descriptions of ThreadSanitizer results for each program in the Concrat Subset benchmark.

Benchmark	ThreadSanitizer result description	Verdict
axel	Two previously reported races: #354: output/progress bar; not fixed. #294: fixed in the latest version.	Race found
C-Thread-Pool	Races on volatile variables.	Race found
dnspod-sr	Hard to test (DNS server).	Not tested
EasyLogger	One race: the unlock should be after writing to `elog` on line 488.	Race found
fzy	None detected (only single query; enough to expose bugs if locks are removed).	No races
klib	One non-atomic read/atomic write violation: the read on line 218 races with the atomic write on line 241; seems harmless.	Race found
level-ip	Hard to test (TCP/IP stack, requires network setup).	Not tested
libfaketime	None detected (the only spawned thread is joined immediately).	No races
lmdb	None detected (low coverage, hard-coded test case).	Low coverage
Mirai-Source-Code	Hard to test (malware).	Not tested
nnn	None detected (low coverage, interactive terminal).	Low coverage
phpspy	Hard to test (monitoring).	Not tested
pigz	None detected (decent coverage).	No races
ProcDump-for-Linux	One mutex destroy/timed wait race: mutex destroyed on line 670 is used in timed wait on line 2093.	Race found
Remotery	Races on volatile variables.	Race found
streem	One non-atomic read/atomic write violation: similar to klib, the read on line 8569 races with the atomic write on line 8695.	Race found
the_silver_searcher	One (pedantic) race: multiple threads may write 1 on line 3886.	Race found
uthash	None detected (uses very non-granular locking).	No races

E Concrat benchmarks' ground truths

The ground truths for the programs in the Concrat are not known. The original paper [43] evaluated a method for translating these programs to Rust's safe locking API, which should reveal races at worst during execution. Their testing did not reveal any races.

We ran ThreadSanitizer (TSan) on all programs in our Concrat Subset using `gcc 13.3.0`. The results are detailed in Table 10. ThreadSanitizer identified data races in eight of the programs. There are two interesting cases worth mentioning. Axel has both a benign race and a data race fixed in subsequent versions. The latter involves the struct-in-array idiom, except the struct is erroneously accessed with the lock of another struct, showing the importance of tracking equalities between integer indices. The other interesting case is the race in EasyLogger because it can be fixed by moving a statement up by a single line; thus, it serves as an excellent use case for incremental verification tools. The other races are likely benign but ideally should be replaced with C11 atomics.

For the programs where we could not elicit a race, there are four programs where we are confident—based on manual inspection and mutating locking operations—that TSan is meaningfully analyzing the programs. The rest of the programs are either networked, interactive, malware, or monitoring, and we encountered the following issues: four could not be tested and two had low coverage.

Our artifact (see Data-Availability Statement) includes the TSan logs and the scripts used to elicit the races.

Table 11. Names of the unsupported library functions for the tools that had "LibFuns" marked in the cells of Tables 2 and 8.

Benchmark	Dartagnan-23	Dartagnan-24	UAutomizer-23	UAutomizer-24	CPAchecker-23
axel	setlocale	__errno_location,	-	-	memset
C-Thread-Pool	puts	__isoc99_sscanf,	-	-	-
dnspod-sr	-	accept, atoi, bind,	-	-	memset
EasyLogger	setbuf	close, connect,	pthread_attr_init	sem_post	-
fzy	fp	daemon,	-	-	-
klib	fprintf	epoll_create,	-	-	-
level-ip	-	epoll_ctl,	pthread_rwlock_init	pthread_rwlock_init	memset
libfaketime	sigemptyset	epoll_wait, fclose,	-	-	-
lmdb	time	fcntl, fflush, fgets,	pthread_getspecific	pthread_getspecific	memset
Mirai-Source-Code	fprintf	fopen, fprintf,	-	-	-
nnn	rawmemchr	getopt, getpagesize,	mbstowcs	mbstowcs	-
phpspy	-	getpid,	-	-	-
pigz	-	gettimeofday,	pthread_getspecific	-	-
ProcDump-for-Linux	openlog	htonl, htons,	-	sem_init	-
Remotery	signal	inet_pton, kill,	sin	pthread_key_create	-
streem	-	listen, llvm.va_end,	log10	pow	memset
the_silver_searcher	-	llvm.va_start,	-	-	memset
uthash	pthread_rwlock_init	mkdir, …	pthread_rwlock_init	pthread_rwlock_init	-

F Details about unsupported library functions

Table 11 details the names of the unsupported library functions of the tools for the Concrat benchmarks. Details for Deagle have been omitted, as its output only indicates "Unsupported library function!" without specifying which functions are unsupported. For CPAchecker, only the details from the 2023 tool are reported, as the 2024 version had no failures related to unsupported library functions. The 2024 version of Dartagnan did not limit itself to specifying one unsupported library function per program; instead, it provided a full list of unsupported functions present in the analyzed program. The Dartagnan-24 column shows the first part of this list for dnspod-sr as an example.

G Analysis of Unsolved Data Race Verification Challenges from SV-COMP

The investigation into the SV-COMP tasks from the NoDataRace category aimed to identify common characteristics in the verification challenges that current tools could not overcome. The competing verifiers implement different technologies, have different strengths and weaknesses, and have different potential for scalability to real-world programs. One would expect different verifiers to succeed at verifying different coding idioms. We thus also investigate the verification results on a more granular basis, studying which tasks were solved (or not) by which verifiers. We considered the 27 tasks with True verdicts ("no races") that none of the tools from Section 4.1 were able to prove correct at SV-COMP, and Table 12 shows our findings. During the manual investigation, it came to light that 16 of these tasks were mislabeled and should actually have a False expected verdict because they contain a data race.[3] This leaves only 11 tasks with True verdicts unsolved by any of the tools.

[3]Our fixes to the verdicts were accepted by the SV-COMP community: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1450, https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1451 and https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1473.

Table 12. SV-COMP 2023 NoDataRace tasks with a True verdict that are unverified by all of the selected tools, along with the causes.

Task	Most prominent cause of failure to verify	Affected tools
01b_inc, 03_incdec,	Simulated locking with boolean variables	Goblint, Locksmith
48_ticket_lock_low_contention_vs	SV-COMP atomics	Deagle
	Unbounded number of threads	CPAchecker, Dartagnan, UAutomizer
bounded_buffer	Array initialization completeness	Goblint
	Bounded thread joining	Locksmith
	Unsupported syntax	CPAchecker, Dartagnan
	Unidentified timeout	UAutomizer
safestack_relacy	Per-thread index from lock-free algorithm	Goblint
	SV-COMP atomics	Locksmith
	Unsupported syntax	CPAchecker, Dartagnan
	Unidentified timeout	Deagle, UAutomizer
workstealqueue_mutex-2	Per-thread index from lock-free algorithm	Goblint
	SV-COMP atomics	Locksmith
	Unsupported library function	Deagle
	Unsupported syntax	CPAchecker, Dartagnan
	Unidentified timeout	UAutomizer
linux-3.13--drivers--* (5 tasks)	Unidentified due to program size	
(16 tasks)	Mislabeled: answer should be False, but had a True expected verdict by mistake	
Total: 27 tasks		

Five unverified tasks were not looked into due to being more than 7,000 LoC each, thus making it difficult to find each tool's most prominent cause of failure. On three other tasks, we conjecture that the tools based on model checking exhaust their resources due to the unbounded number of threads, an idiom that they do not handle [10, 27]. On the same three tasks, however, Goblint and Locksmith failed due to locking being simulated using boolean variables. On the remaining three tasks, multiple tools did not support the syntax or the used library functions. Despite using a small fixed number of threads, there were still timeouts whose causes could not be determined from the tools' output. In bounded_buffer, Locksmith fails to handle the joining of threads [62], while Goblint cannot verify that an array is sufficiently initialized by the time of use. Two tasks implement lock-free data structures via SV-COMP-specific atomic operations. Goblint cannot perform the delicate value analysis required to verify them.

The leading tool, Goblint, could not verify an additional 26 tasks. Goblint primarily struggles with verifying tasks involving value-based locking schemes that require tracking precise interleavings between a finite number of threads. These are generally handled well by tools using model checking.

The second-best tool, Deagle, uses Goblint internally and thus should fail on fewer tasks than Goblint. However, it received a lower score than Goblint due to superficial implementation issues. For example, it first attempts to unroll loops for its bounded model checker, and if this process fails, it does not run Goblint. Alternately, it may run Goblint, which returns zero potential races, but then Deagle gives an Unknown verdict due to unsupported library functions.

The third-ranking tool, Dartagnan, produces 181 errors, which rarely refer to the offending constructs in the analyzed program; 21 errors because it does not support threads created in a loop; and 45 timeouts.

Table 13. Descriptions of the Joern queries used in Section 5.2 to count occurrences of syntactic properties that indicate the use of the identified challenging implementation idioms.

#	Relevant Property	Query description
1	Thread creation within a looping construct	Find all transitive callers of `pthread_create`, including `pthread_create` itself. Count benchmarks where a call to one of these functions is a node of a `while` or `for` AST.
2	Flow from memory allocation to thread creation	Count benchmarks with `pthread_create` calls where the 4th argument `arg` is reachable from a call to `malloc`, `alloca` or `calloc`.
3	Giving an integer or indexed address to created thread	Count benchmarks with `pthread_create` calls where the 4th argument `arg` contains: • a (pointer) addition operation or an indexing operation, • an integer variable that also occurs in the condition of a parent looping construct.
4	Use of thread-local storage (`__thread`)	Count benchmarks that include the annotation `__thread`.
5	Use of Posix API for thread-local data (`pthread_key`)	Count benchmarks with a variable of the type `pthread_key_t`.
6	Joining threads with array indexing	Count benchmarks where the first argument to `pthread_join` contains a (pointer) addition operation or an indexing operation.
7	Thread creator has control dependency on shared int	Count the benchmarks in which a method that calls `pthread_create`, also uses a global `int` variable within a condition of a control statement.
8	Atomics (`__sync` or `__atomic`)	Count the benchmarks where `__sync` or `__atomic(.*fetch|store|load|exchange)` functions are used.
9	Semaphores (`sem_wait`)	Count the benchmarks where `sem_wait` is used.
10	Use of `pthread_cond`	Count benchmarks that call `pthread_cond.*`.

Similarly, we found the fourth-ranking tool, Ultimate Automizer, difficult to judge because its inability to prove correctness is due to timeouts or running out of memory. For some classes of tasks, it verifies one variation slightly under the time limit but exhausts the limit on another variation. While there are some obvious constructs, such as creating 10,000 threads, that cause the Ultimate family of tools to run out of memory, there are also cases that only involve two threads where it fails to reason about dynamically allocated locks.

H Descriptions of the Joern queries for prevalence analysis

The Joern queries used to collect data in Table 5 are detailed in Table 13. The code for these queries is included in the accompanying artifact (see Section 8).

We validated the correctness of our queries by running them on the extracted micro-benchmark set. The results, detailed in Table 14, confirm that the syntactic properties associated with each idiom in Section 3 are accurately identified in the corresponding micro-benchmarks.

Table 14. Syntactic properties identified in our extracted micro-benchmarks using the Joern queries.

Relevant Property	Fig.	1	2	3	4	5	6	7	8	9	10
per-thread-struct	1a	✓	✓				✓				
per-thread-array-ptr	1b	✓	✓	✓			✓				
per-thread-array-index	1c	✓		✓			✓				
per-thread-array-init	1d	✓	✓	✓			✓				
per-thread-struct-in-array	-	✓	✓	✓			✓				
per-thread-index-inc	1e	✓					✓				
per-thread-index-bitmask	-	✓					✓				
thread-local-value	2a	✓			✓		✓				
thread-local-value-cond	-	✓			✓		✓				
thread-local-value-dynamic	-	✓			✓		✓				
thread-local-pthread-value	2b	✓				✓	✓				
thread-local-pthread-value-cond	-	✓				✓	✓				
thread-join-array-dynamic	3a	✓					✓				
thread-join-array-const	-	✓					✓				
thread-join-binomial	-	✓		✓			✓				
thread-join-counter-outer	3b	✓						✓			✓
thread-join-counter-inner	-	✓						✓			✓
atomic-gcc	4a	✓					✓		✓		
semaphore-posix	4b	✓					✓			✓	
value-barrier	4c	✓					✓				✓