Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation

Oskar Haarklou Veileborg
Aarhus University
Denmark
oskar@cs.au.dk

Georgian-Vlad Saioc
Aarhus University
Denmark
gvsaioc@cs.au.dk

Anders Møller
Aarhus University
Denmark
amoeller@cs.au.dk

ABSTRACT

Channel-based concurrency is a widely used alternative to shared-memory concurrency but is difficult to use correctly. Common programming errors may result in blocked threads that wait indefinitely. Recent work exposes this as a considerable problem in Go programs and shows that many such errors can be detected automatically using SMT encoding and dynamic analysis techniques.

In this paper, we present an alternative approach to detect such errors based on abstract interpretation. To curb the large state spaces of real-world multi-threaded programs, our static program analysis leverages standard pre-analyses to divide the given program into individually analyzable fragments. Experimental results on 6 large real-world Go programs show that the abstract interpretation achieves good scalability and finds 104 blocking errors that are missed by the state-of-the-art tool GCatch.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Static Program Analysis, Concurrency, Channels

ACM Reference Format:
Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. 2022. Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3561154

1 INTRODUCTION

The key feature of the Go programming language is its channel-based approach to concurrency with lightweight threads. Instead of communicating via shared memory, Go advocates the use of channels to avoid errors involving data races. This language design choice is a central reason for the popularity of the language. More than 270 000 open source projects on GitHub use Go, including prominent applications, such as, Docker and Kubernetes. However, using channels does not eliminate all concurrency-related errors. Previous work has shown that channel-related concurrency errors are frequent in Go programs [13, 14, 24]. A common erroneous pattern involves a thread waiting on a blocking channel operation that will never be unblocked by any other thread, often due to some unexpected condition occurring for potential communication partners. Repeated occurrences of such progress failures may drain system resources and eventually cause execution to halt.

To detect such blocking errors automatically ahead of program execution, we propose an approach based on abstract interpretation. Our analysis first locates channel creation and communication operations by leveraging existing basic analyses for producing call graphs and aliasing information, and identifies for each channel a program fragment covering its operations. Each fragment is then analyzed separately using abstract interpretation, hence the term *localized abstract interpretation*, to infer a finite transition system that flow-sensitively models the state space for the relevant threads and channels in the fragment. Dynamic thread creation may lead to a statically unbounded number of threads, causing challenges for static analysis. The localized analysis approach partially circumvents this issue for fragments that may execute an unbounded number of times at run-time but where each execution only involves a bounded number of threads. The last step consists of analyzing the generated transition systems, looking for configurations where a thread may be blocked indefinitely, in which case a potential error is reported.

Our approach is inspired by GCatch [14], which aims to detect the same category of errors and similarly analyzes selected program fragments one-by-one. The central difference is that GCatch does not use abstract interpretation but instead enumerates potential execution paths for each thread within the fragment and uses an SMT solver to compose the paths and detect blocked threads. We believe it is simpler to model channel operations and other Go language constructs using abstract interpretation instead of SMT encodings. The tool Gomela [5, 6] has similar goals and is based on bounded model checking of Promela-encoded Go programs. It obtains high scalability on real-world code, but detects relatively few blocking errors. Another recent approach is GFuzz [13], which detects blocking errors by fuzzing concrete executions obtained by running the programs' test suites. That approach relies on high-coverage test suites to be effective.

Preliminary experiments show that our localized abstract interpretation approach scales to large Go programs (typically analyzing each fragment in less than a second), it is capable of finding many blocking errors that are missed by the existing state-of-the-art tools, and it has an acceptable false positive rate (less than 50%).

These results are enabled by some pragmatic design choices: (1) A common programming pattern involves worker pools with

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3561154

correlated loops, which we handle in a light-weight manner by modeling only a fixed number of loop iterations. (2) The localized abstract interpreter models a coarse-grained thread scheduler that intuitively ignores potential shared-memory data races, giving a substantial state space reduction. As our goal is not to detect data races but channel-related issues, it is acceptable that not all possible interleavings are explored. Disabling these techniques reduces the performance and accuracy of the analysis.

In summary, the contributions of this paper are:

- We demonstrate that localized abstract interpretation is a promising approach to detect channel-related concurrency errors. This is notable, because abstract interpretation is historically rarely used for reasoning about such concurrency errors due to the large state spaces that often appear.
- By experimentally evaluating the approach on 6 large real-world Go programs, we show that it compares favorably with the state-of-the-art tools GCatch and GFuzz. Specifically, it detects 104 blocking errors that GCatch misses and 84 that GFuzz misses (thereof 76 missed by both), and most program fragments are analyzed in seconds. Additionally, we show that the pragmatic design choices are important for the efficacy.
- We report on typical scenarios that cause false positives and false negatives, which suggests interesting opportunities for future work.

The approach is implemented in the tool Goat.[1]

2 BACKGROUND

Go is a statically-typed concurrent imperative language geared towards systems development. Threads (called "goroutines") can be created dynamically like in many other languages. Although Go supports traditional shared-memory communication among threads using locks and other basic synchronization mechanisms, its hallmark concurrency feature is channels inspired by Hoare's CSP. A channel is a bounded queue that can be accessed by multiple threads. Reading from a channel blocks until data is available, and sending to a channel blocks if the channel is full. The channel capacity is selected when the channel is created. For example, `make(chan T, 3)` creates a channel for values of type `T` with capacity 3. Channels with capacity zero are called *synchronous* because they require reads and corresponding writes to happen simultaneously. Alongside the send and receive operations (`c <- ···` denotes writing to a channel `c` and `··· <- c` denotes reading), Go's `select` statement allows nondeterministic choice among enabled channel operations. Channels may also be closed, at which point their blocked receive operations are unblocked and further send or close operations will fail.

Channel-based concurrency is a powerful and popular language mechanism that prevents low-level data races, but it does not prevent all concurrency issues. If *all* threads are blocked on some channel operations, waiting for other threads to send or receive, Go's built-in deadlock detector aborts execution. However, a much more common situation is that some but not all the threads are blocked, waiting for channel operations that can never occur, because some thread has taken an execution path that was not anticipated by the programmer. This situation may violate desirable progress properties of the program, and it causes "goroutine leaks" that consume precious system resources. The goal of our work is to automatically detect whether such blocking errors are possible in a given Go program.

```
1   type Server interface { // Server API
2     Error() chan error
3     Ready() chan struct
4   }
5
6   func NewServer(cfg) Server {
7     s := &server{ // Server object
8       ...
9       readyc: make(chan struct{})  // Ready channel
10      errc:   make(chan error, 16) // Error channel
11    }
12    ...
13    _, err = net.Listen(s.from.Scheme, addr)
14    if err != nil { // Server startup error
15      s.errc <- err
16      s.Close()
17      return s
18    }
19    go s.listenAndServe()
20    return s
21  }
22
23  type server struct { // Server data structure
24    ...
25    readyc chan struct{}
26    errc   chan error
27  }
28
29  func (s *server) listenAndServe() {
30    ...
31    close(s.readyc) // Close ready channel
32    for { ... } // Listen-and-serve loop
33  }
34
35  func (s *server) Ready() { // Ready channel getter
36    return s.readyc
37  }
38  func (s *server) Error() { // Error channel getter
39    return s.errc
40  }
41
42  func testServer() {
43    ...
44    s := NewServer(cfg)
45-   <-s.Ready() // Potential blocking error
46+   select {
47+     case <-s.Ready(): ... // Proceed normally
48+     case err := <-s.Error(): ... // Handle error
49+   }
50    ...
51  }
```

Figure 1: A blocking error in *etcd*. (Irrelevant details have been elided, and explanatory comments have been added.)

Figure 1 illustrates such an error, found in *etcd*,[2] a distributed key-value store implemented in Go. It involves creating a configurable server (line 44), and waiting until it is ready (line 45). Servers implement the `Server` interface (line 1), which contains a

[1] **Go a**nalysis **t**ool

[2] https://etcd.io/

`Ready` method that returns a channel. Since creating a server may be asynchronous, a rendezvous point is established by reading from this channel. The servers produced by `NewServer` (line 6) are represented by the `server` structure (line 23), where the `Ready` method (line 35) returns the channel embedded in the `readyc` field. Servers are ready when the `readyc` channel is closed (line 31). `NewServer` normally achieves this by spawning a thread (expressed using the **`go`** keyword), **`go`** `s.listenAndServe` (line 19), that executes the **`close`** operation. However, if an error occurs (line 14), the function returns without closing the channel. In this case, reading from the `readyc` channel will block indefinitely. The proposed fix for this blocking error is to read from both the `readyc` and `errc` channels simultaneously using a **`select`** statement (lines 45–49).

This example illustrates how channels may easily be misused. Any client using the API similarly to the `testServer` function in Figure 1 exposes itself to the resource leak. Fixing the error requires knowledge of the implementation of `NewServer` and `Ready`, which is complicated by having the implementation of `*server` methods hidden behind the `Server` interface.

This particular error is missed by the existing tools GCatch and GFuzz. GCatch fails to detect the erroneous execution path through the relevant part of the program. GFuzz performs fuzzing of program executions by re-ordering choices of case clauses in **`select`** statements, which does not suffice to discover that `err` may be non-**`nil`** in this case. In contrast, GOAT successfully discovers the error by analyzing a program fragment containing the functions that involve operations on the `readyc` channel. The fragment does not contain the function `net.Listen`, so the analysis makes a worst-case assumption about the possible value of `err`, therefore considering the path with the server start-up error where no thread is spawned at line 19. The developers of *etcd* have subsequently confirmed the error and approved the proposed fix.

3 APPROACH

Reasoning automatically about the presence or absence of blocking errors in Go programs requires flow-sensitive analysis of multi-threaded code and channel state. Our approach to obtain scalability to large, real-world programs is to consider each syntactic channel creation site individually (e.g., line 9 in Figure 1) and ignore program code that is unlikely to affect whether operations on channels created there may block. As an example, the code involving the blocking error of the `readyc` channel, and the suggested fix shown in Figure 1 is a tiny fraction of the 180 KLOC that constitute *etcd*.

Overall, the analysis of a given Go program is divided into three main phases:

(1) The **pre-analysis** phase identifies channel creation sites syntactically, and for each of them selects a *program fragment* consisting of functions that likely cover the relevant channel operations. To this end, we leverage the existing Go parser and Andersen-style points-to analysis [22]. The selection of program fragments is inspired by the one used by GCatch [14] as explained in detail in Section 3.1.
(2) The main work is performed in the **abstract interpretation** phase that analyzes each program fragment and builds a *super-location graph*, which is a finite transition system that models the state space of the fragment. In Section 3.2 we describe the abstract domain and abstract semantics of this analysis, and how it handles interactions with program code outside the fragment. These abstractions are carefully designed to track enough information to enable reasoning about blocking channel operations, while allowing typical fragments to be analyzed in less than a second.
(3) In the **blocking error detection** phase described in Section 3.3, the superlocation graphs are traversed, searching for abstract threads at channel operations with no possible unblocking path. Such threads may likely block indefinitely at run-time.

We conjecture that most uses of channels in real-world Go programs are amenable to such localized analysis. By bounding the analysis time for each program fragment, we effectively obtain an analysis technique that scales linearly in the size of the program (if ignoring the time for the pre-analysis). For programs with multiple entry points, the precision of the pre-analysis can be improved by running it separately for each entry point, essentially treating each entry point as a separate program.

3.1 Pre-Analysis

The pre-analysis first runs the ***points-to analysis*** available in the `pointer` package developed by the Go team [22]. Since channels are first-class values in Go, the resulting points-to information [1] tells us for each channel creation site which channel operations may involve channels created at that site. Continuing the *etcd* example from Figure 1, channels created at line 9 may be used only at lines 31 and 45 (before the fix is applied). We henceforth identify channels by their syntactic creation site in the program. The points-to analysis result also includes a call graph that approximates which functions may be called at each call site.[3] To support the next steps, we compute the strongly connected components (SCCs) of the call graph and organize them in reverse topological order.

With these initial steps in place, we can perform ***fragment construction*** that selects a set of functions, called a *fragment*, for each channel c in the program. This process is inspired by GCatch [14] but with some extensions as explained below. The basic idea is to locate functions that are likely relevant for reasoning about operations on c, building on the points-to and call graph information. Since this may involve other channels, we first identify a set of likely relevant channels, $\mathcal{P}(c)$ (called P_{set} in [14]). Intuitively, communication operations on channels not in $\mathcal{P}(c)$ are ignored by the abstract interpretation of the fragment constructed for c. The set $\mathcal{P}(c)$ consists of the channel c itself and any other channel c' that satisfies one or both of the following conditions:

C1: c' and c are mutually dependent. Channel c depends on c' if an operation on c that may unblock another operation on c is intra-procedurally reachable from a blocking operation on c'.
C2: c' and c are used in different **`case`**s of the same **`select`** statement.

Figure 2a illustrates C1 by a blocking error that is due to a communication mismatch, which would not be revealed by analyzing channels individually. We have that `a` depends on `b` because the

[3] Imprecision of this existing analysis can lead to a large number of callees for some call sites, especially where interface method invocation is involved. To prevent such situations from affecting the main analysis, we prune the call graph at sites where the number of callees exceeds an arbitrarily limit (10 in our experiments). This pragmatic choice increases precision and speed, at the cost of causing unsoundness in the analysis.

read operation on line 55, which may unblock the write on line 57, intra-procedurally requires reading from `b` on line 54. Conversely, `b` depends on `a` because writing to `b` on line 58 requires writing to `a` first on line 57. Restricting to intra-procedural reachability is a heuristic that prevents call graph imprecision to lead to very large $\mathcal{P}(c)$ sets.

Figure 2b motivates C2 by showing an example of a blocking `select` that would not be detected if $c' \notin \mathcal{P}(c)$ and $c \notin \mathcal{P}(c')$.

An additional condition for both C1 and C2 is that the dominator of c' is reachable from the dominator of c in the call graph. Here, the *dominator* of a channel is the dominator in the call graph of all the functions that might create, use, or return the channel. This additional condition helps limiting the sizes of the $\mathcal{P}(c)$ sets, thereby providing a more fine-grained analysis of the program.

We extend the definition of $\mathcal{P}(c)$ beyond the P_{set} construction of GCatch by also including any channel c' in $\mathcal{P}(c)$ that satisfies the following condition:

C3: c' might carry c as a payload, irrespective of how their dominators are related.

This addition of C3 is owed to our empirical observation of programming patterns involving channels with channel payloads. Figure 2c illustrates a non-blocking example where `a` would not be included in $\mathcal{P}(\texttt{b})$ by GCatch, leading the analysis to lose track of the connection between `b` and the payload of `a` when ignoring operations on `a`. Including `a` in the set of channels relevant to `b` allows precise analysis of all communication in the fragment.

For each set $\mathcal{P}(c)$ we now define a program fragment $\mathcal{F}(c)$ containing each function that for some $c' \in \mathcal{P}(c)$ either creates, returns, or performs a communication operation on c'. Additionally, $\mathcal{F}(c)$ includes all ancestors in the call graph, up to the dominator of those functions. We denote this dominator as the *fragment entry*. Note that $\mathcal{F}(c)$ typically does *not* include all functions that may be called from the fragment. Intuitively, excluded functions that may be called from within the fragment are considered irrelevant to the operations on c. For the *etcd* example in Figure 1, the program fragment obtained for the channel created at line 9 contains all the functions defined in the figure, and its entry is one called `testServer`.

Next, a ***side-effect analysis*** is performed, bottom-up in the SCCs. A function is marked as potentially inducing side-effects to a points-to analysis allocation site if the function itself, or any function it may call directly or transitively, might write to that allocation site. This information, derived from the points-to and call graph analysis, is used in the abstract interpretation phase for estimating the potential side-effects of function calls outside the fragment under analysis.

In the *etcd* example, the call to `s.Close` at line 16 is not relevant for exposing the bug involving `readyc` and the function being called is not included in the fragment for the `readyc` channel. A naive over-approximation of possible side-effects of that function would assume that the `readyc` and `errc` fields may be overwritten by invoking `s.Close`, losing the guarantee that future reads of these fields yield the references to the initial channels (lines 9–10). The side-effect analysis prevents this by marking the invocation of the `Close` method as not overwriting these fields of `s`.

The results of the pre-analysis phase are used during abstract interpretation, described in the next section.

```
52  a, b := make(chan int), make(chan int)
53  go func() {
54    <-b
55    <-a
56  }()
57  a <- 1
58  b <- 2
```

(a) Blocking error resulting from channel communication mismatch, captured by condition C1.

```
59  a, b := make(chan int), make(chan int)
60  select {
61    case <-a:
62    case <-b:
63  }
```

(b) Blocking select statement, captured by condition C2.

```
64  a := make(chan chan int, 1)
65  go func() {
66    b := make(chan int)
67    a <- b
68    b <- 3
69  }()
70  <-<-a
```

(c) Common non-blocking communication pattern involving channels with channels as payload, captured by condition C3.

Figure 2: Motivating examples for constructing $\mathcal{P}(c)$.

3.2 Localized Abstract Interpretation

This section gives an overview of the abstract interpretation phase by first defining the analysis domain and then describing the abstract semantics in the context of analyzing program fragments. The analysis is designed such that it aborts if it encounters certain difficult situations, which helps ensure a good balance between analysis time, precision, and recall when analyzing a given fragment.

Analysis domain. The abstract domain is a complete lattice that models program behavior flow-sensitively. It is defined as

$$\mathcal{A} = \overbrace{(\mathcal{G} \hookrightarrow \underbrace{(N \times F)}_{\text{control locations}})}^{\text{superlocations}} \to \overbrace{(\mathcal{L} \to \mathcal{V})}^{\text{abstract states}}$$

representing bindings from superlocations (defined below) to abstract states. We next explain each of the components, N, F, $\mathcal{G}$, $\mathcal{L}$ and $\mathcal{V}$.

Using the `ssa` package [23] and the call graph from the pre-analysis, we obtain a control flow graph (CFG) for the given program with a set of nodes N and functions F.

The set $\mathcal{G}$ represents abstract threads. We define $\mathcal{G}$ as the set of `go` instructions that appear in the program, i.e., $\mathcal{G} \subseteq N \cup \{\epsilon\}$, where the special element ϵ represents the main thread. Intuitively, each thread that may appear at run-time is represented abstractly by the `go` instruction where it was spawned.

The domain of $\mathcal{A}$ is the set of *superlocations*, where each abstract thread from $\mathcal{G}$ is bound to a *control location* (or is undefined). In a control location $(n, f) \in N \times F$, the CFG node n represents the next instruction to be executed by the corresponding thread. The function f is the one where the thread started execution. For example,

for threads created at line 19 in Figure 1, f is the `listenAndServe` function defined at lines 29–33. Intuitively, for a superlocation where $g \mapsto (n, f)$, thread $g \in \mathcal{G}$ is currently at node n and terminates when it leaves f. If g does not map to any control location, it means that no corresponding thread exists. Since control locations are derived from control flow nodes, any (n', f) is a successor of (n, f) if n' is a successor of n in the CFG. Additionally the control location (f_{exit}, f) where f_{exit} is the exit CFG node for f, indicating that a thread is at the exit of the function where it started, has a special successor $(\circ, f)$, indicating that the thread has terminated.

$\mathcal{L}$ is the set of abstract stack and heap locations. These are identified syntactically by the variable declaration sites and allocation sites, respectively, and are further distinguishable by the abstract thread that allocates them (akin to context-sensitive heaps, or heap cloning, in context-sensitive points-to analysis [20]).

$\mathcal{V}$ is the domain of abstract values. It is a product lattice that combines standard analysis domains for each Go type: constant propagation for basic primitive types, points-to sets for reference types (pointers, interfaces, channel locations,[4] closures, references to built-in dynamic data structures), and a map lattice for aggregate data types, e.g., `struct` values. This is extended with the domain of *abstract channels*, which is a product of four sub-domains:

(1) The channel status, represented by a four-element lattice consisting of *OPEN*, *CLOSED*, undefined ($\bot$), and unknown ($\top$). This information is relevant for modeling channel communication (closed channels do not block on reading, whereas sending produces an error) and payload data flow (closed channels with empty buffers produce the zero-value for the payload type).
(2) The capacity lattice, which is a constant propagation lattice for natural numbers, keeping track of the channel capacity.
(3) The current buffer size lattice, as an interval lattice bounded in height by the number encoded in the capacity, if statically decidable, or a one element lattice (unknown) otherwise.
(4) The channel payload, which is itself the abstract value lattice. Possible payload values are joined for channels with capacities greater than 1. For the example in Figure 2c, this definition allows the precise modeling of the payload of channel `a` as the reference to channel `b`.

While the abstract domain of channel payloads makes the value lattice inductive, the type system of Go ensures that the height of this lattice is always finite for any given Go program: named types may not have cyclical definitions, except via indirection, which is modeled by points-to sets.

Figure 3 depicts the execution paths of the example in Figure 1 as a graph. The top of each node represents a superlocation,[5] and the bottom is the associated abstract state. For example, at the superlocation $[\epsilon \mapsto$ `if err != nil`$]_2$, the channel `readyc` has been previously initialized, and its status is *OPEN* (we here focus on the channel status and omit all the other information being represented by the abstract states). At $[\epsilon \mapsto$ `<-s.Ready()`$, g_1 \mapsto$ `close(s.readyc)`$]_6$, we model the configuration where the main thread waits on `<-s.Ready()`, and the child thread denoted g_1, which

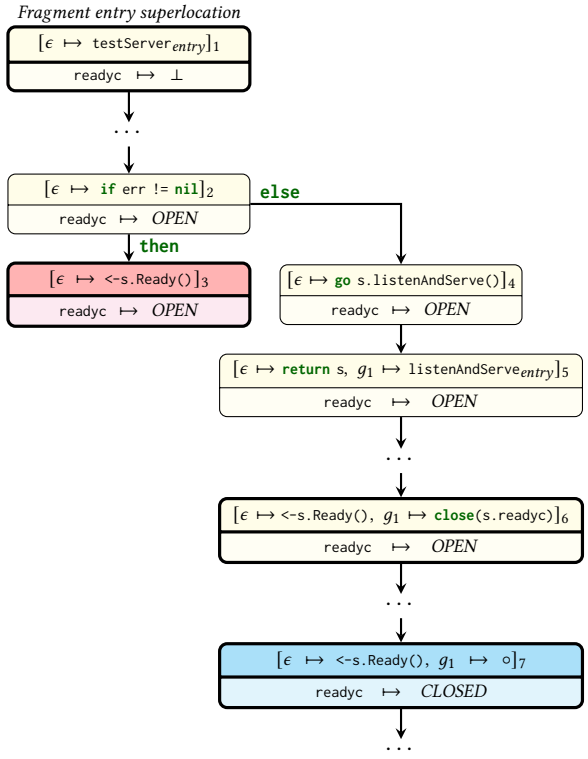

Figure 3: Subset of the control flow in the example from Figure 1, where superlocations are paired with abstract states.

is created at line 19, will next close the `readyc` channel. In abstract states of successors of this superlocation, `readyc` is *CLOSED*. The edges in the graph express the successor relation between the superlocations, and the nodes with thick borders constitute those included in the superlocation graph as explained later. For an execution that takes the 'then' branch, reading from channel `readyc` (at the node marked with red) will block because there is no communication partner. Conversely, on the 'else' branch, g_1 closes `readyc`, unblocking the read operation in the main thread (at the node marked with blue).

Notice that the abstract domain of the analysis has been designed such that it maintains an abstract state for each superlocation rather than tracking state for individual threads. In the example, the status of the `readyc` channel importantly depends on the combination of where the different threads are in the program.

Abstract semantics. Given a program fragment $\mathcal{F}(c)$ with entry f, the abstract interpretation computes an element of the abstract domain $\mathcal{A}$ using a fixpoint computation, as usual in monotone frameworks [8].

Analysis is initiated by assigning an initial abstract state to the entry superlocation $[\epsilon \mapsto (f_{entry}, f)]$, i.e., the superlocation that only maps the main thread, ϵ, to the entry of the function f, where f_{entry} is the entry CFG node for f. All other threads are inactive (i.e., their control locations are undefined) at the entry superlocation. Since the fragment entry is generally not a program entry point, the initial abstract state is constructed such that it conservatively

[4] To account for potential aliasing, channels are treated as a special kind of objects along with other reference types.
[5] We denote a CFG node by its syntax and omit the function component of control locations for brevity. The subscript labels, e.g., $[\ldots]_1$, uniquely identify the superlocations.

```
71  for i := range list {
72    go func() {
73      ...
74      if ... {
75        ch <- res
76      }
77    }()
78  }
79  for i := range list {
80    ...<-ch
81  }
```

Figure 4: Example program where communication involves correlated loops.

models the values of any parameters or free variables using the $\top$ lattice element for the corresponding Go type. To reduce the size of the abstract state, reference types are handled lazily as explained below (see 'Localized analysis').

The abstract interpretation repeatedly applies the transfer functions for the next Go instruction for each active thread, using a worklist algorithm until the least fixpoint is reached. For brevity, we omit a detailed description of the transfer functions; intuitively, they simply model the semantics of Go instructions in the control flow graphs obtained via the `ssa` package [23], according to the abstraction established by the analysis domain. Some interesting analysis design choices are involved, however, in enabling strong updates of control locations, modeling the thread scheduler, and handling calls to code outside the fragment being analyzed, as explained next.

Enabling strong updates of control locations. To provide sufficient analysis precision about channel communication, it is important that each abstract thread represents at most one run-time thread. This property makes it possible to strongly update the CFG node to its successors when processing its transfer function for an abstract thread.[6] If the analysis at a superlocation encounters a `go` instruction that already represents an abstract thread (meaning that a control location is already assigned to that abstract thread), it simply aborts. In complete executions of whole programs, it is very common that a syntactic `go` instruction is encountered multiple times. However, because we analyze not whole programs but relatively small program fragments, the consequences are less severe, as many `go` instructions encountered by the analysis will spawn functions that are not in the analyzed fragment, making the analysis simply ignore them (while treating their potential side-effects conservatively). For example, the fragment of `readyc` in Figure 1 may be analyzed independently, even if the fragment entry, `testServer`, may be reached an unbounded amount of times in a complete program execution.

Handling correlated loops. Figure 4 shows a common pattern in Go programs. At lines 71–78, a statically unknown number of worker threads are created to compute some values that are then sent to the channel `ch`. At lines 79–81, the results are collected by the main thread. Assume that, in normal executions, the worker threads always take the branch to the send operation at line 75. In this situation, the total number of send operations is equal to the total number of receive operations, so all the threads eventually progress. However, if one of the worker threads does not execute its send operation, the main thread is blocked indefinitely. This program exemplifies correlated loops, where two loops perform the same number of iterations as determined by the number of elements in `list`. We take a lightweight approach to handling this pattern by assuming that all loops perform exactly one iteration. This has the advantage of reducing the number of analysis aborts triggered by the conditions discussed above, while allowing the analysis to detect blocking errors as the one in Figure 4. If the error in that example is fixed by having the send operations occur unconditionally (e.g., by removing line 74), the analysis reports no error.

Modeling the thread scheduler. Although channels are the recommended mechanism in Go for communicating between threads, the language also supports shared-memory concurrency. The goal of our analysis is not to detect data races but blocking errors caused by channel miscommunication.[7] Nevertheless, the possibility of data races means that a perfectly sound analysis would have to consider all possible interleavings of thread executions, at the level of individual instructions, which quickly leads to a combinatorial explosion in the size of the explored superlocation set.

We alleviate this issue by treating non-communicating instruction sequences as if they were atomic, while preserving the interleavings of channel operations. Intuitively, the abstract interpreter only models thread switches that occur when the currently executing thread is ready to communicate (or has terminated), and communication only occurs when all threads are ready to communicate (or have terminated). Such a coarse-grained modeling of thread scheduling substantially reduces the state space and can only result in missed bugs in the presence of race conditions.

To express this more precisely, we classify each control location as either *communicating* or *silent* to denote whether the operation at its control flow node is a communication operation or not, respectively. Each thread in a superlocation is similarly classified, depending on the type of control location it is bound to. At a *silent superlocation*, at least one thread is silent, and at a *communicating superlocation*, all threads are communicating or terminated.

The analysis computes the least fixpoint $a \in \mathcal{A}$ of the analysis constraints by a series of approximants $\bot = a_0 \sqsubset a_1 \sqsubset a_2 \sqsubset \cdots \sqsubset a_n = a$ using a traditional worklist algorithm on superlocations [3, 8]. Let ϕ be a superlocation and σ its corresponding abstracting state. Processing ϕ in the worklist algorithm produces a set of transitions, each consisting of a successor superlocation ϕ' and an updated abstract state σ'.

- If ϕ is communicating, we model *inter-processual* data flow. For each enabled communication operation of each thread, the outgoing transitions are computed. Let g be a thread in ϕ where the instruction at the corresponding control location $\theta = \phi(g)$ is an enabled communication operation according to the abstract semantics of the instruction and the abstract state σ. For every ϕ' obtained by modeling the instruction at θ in ϕ, we have that $\theta' = \phi'(g)$ is a successor of θ (and similarly for any g' which

[6] This notion of strong updates is reminiscent of the one used in points-to analysis [2]. If one abstract thread could represent multiple concrete threads, the CFG node of the control location could only be updated weakly, which would lose the effect of flow sensitivity.

[7] This approach is supported by the design philosophy of the Go design team: *"Don't communicate by sharing memory; share memory by communicating"* [21].

```
82 type S struct { ch chan int; val int; flag bool }
83
84 func entry() {
85   s := S{ch: make(chan int, 1), val: 10, flag: false}
86   init(&s)
87   s.ch <- s.val
88 }
89
90 func init(s *S) {
91   if s.flag {
92     s.val = 0
93   }
94 }
```

Figure 5: Localized analysis example.

is chosen as a potential communication partner, in the case of channel synchronization). Similarly, the corresponding abstract state σ' is obtained by appropriately updating σ according to the abstract semantics of the instruction at θ.

- If ϕ is silent, we model *intra-processual* data flow, by only producing transitions for the next silent thread. The next silent thread is selected by imposing an arbitrary but deterministic order on threads. Let g be the next silent thread of ϕ, and let $\theta = \phi(g)$. The set of outgoing transitions at ϕ is now computed according to the abstract semantics of the (non-communicating) instruction at g relative to the abstract state σ.

In both cases, the next approximant is computed as the least upper bound $a_{i+1} = a_i \sqcup [\phi'_1 \mapsto \sigma'_1] \sqcup \ldots \sqcup [\phi'_k \mapsto \sigma'_k]$ where each pair ϕ'_j, σ'_j for $j = 1, \ldots, k$ is one of the generated transitions.

As an example, since $[\epsilon \mapsto \texttt{return}\ \texttt{s}, g_1 \mapsto \texttt{listenAndServe}_{entry}]_5$ in Figure 3 is silent, we apply the intra-processual analysis, first by modeling the sequential operations of ϵ until `<-s.Ready()` is reached, and then those of g_1 until `close(s.readyc)` is reached (the intermediary steps are elided in the figure). This leads to $[\epsilon \mapsto$ `<-s.Ready()`$, g_1 \mapsto$ `close(s.readyc)`$]_6$, which is a communicating superlocation, where the analysis models possible progress for each enabled operation of ϵ and g_1 via inter-processual data flow. As reading from `readyc` is not enabled, due to `readyc` being *OPEN* and ϵ not having a communication partner, only g_1 can proceed by closing `readyc`. Starting at the successor, we again apply the intra-processual analysis to g_1 (elided), reaching communicating superlocation $[\epsilon \mapsto$ `<-s.Ready()`$, g_1 \mapsto \circ]_7$. At this point, g_1 is terminated, while ϵ can proceed by reading from `readyc`, which is now enabled since `readyc` is guaranteed to be closed.

Localized analysis. The fragment being analyzed may contain calls to code outside the fragment. Unknown primitive values (integers, strings, etc.) that originate from such code are modeled conservatively using the $\top$ lattice element for the corresponding type (representing all possible values of that type), as in the construction of the initial abstract state. Code outside the fragment may also affect the abstract state due to side-effects when references escape the fragment. Heap locations that may be affected according to the side-effect analysis (Section 3.1) and are of a primitive type are similarly overwritten by $\top$ elements for the corresponding type. For every escaping heap location with a reference type, the points-to pre-analysis provides a conservative points-to set that models all possible side-effects to that location.

Unfortunately, Go's standard points-to analysis that we rely on does not support points-to queries to arbitrary heap locations but only to SSA registers. For this reason, our implementation queries the points-to analysis lazily, when the points-to sets of interest reach registers. This also has the effect of reducing the sizes of the points-to sets in the abstract states. However, it causes complications when lazily evaluated points-to sets themselves escape the fragment being analyzed. When that occurs, we pragmatically choose to simply ignore side-effects involving those references. Also, we let the analysis of a fragment abort if a lazily evaluated points-to set contains a reference to a channel in $\mathcal{P}(c)$, as the analysis has likely lost too much precision in that situation.

For the fragment created for the channel allocated on line 85 in the example program in Figure 5, the fragment entry is the function `entry` and the function `init` is not included in the fragment. When the analysis reaches the call to `init`, the reference to `s` escapes the fragment. At this point the abstract state for the object stored at that abstract location corresponding to `&s` is $[$`ch` $\mapsto \{$`make(chan int, 1)`$^{\epsilon}_{85}\}$, `val` $\mapsto$ 10, `flag` $\mapsto$ `false`$]$. Here, `make(chan int, 1)`$^{\epsilon}_{85}$ denotes a channel allocation on line 85 by the thread ϵ. The side-effect analysis tells us that the field `val` may be overwritten in the call to `init`, therefore the abstract value for the field `val` is updated to $\top$. We can soundly keep the abstract points-to set for the field `ch` and the constant `false` as the abstract value for the field `flag`, as these fields cannot be written to in `init`. Consequently, the abstract interpreter knows that the only channel that is operated on at line 87 is `make(chan int, 1)`$^{\epsilon}_{85}$.

To illustrate the technical issue with channel references in lazily evaluated points-to sets, assume we modify the assignment `s.val = 0` on line 92 such that `s.ch` is instead assigned a newly allocated channel. The abstract value for the `ch` field would then be replaced by a lazily evaluated points-to set when the call to `init` is encountered. After the call, when the channel receive operation is reached on line 87, the lazily evaluated points-to set reaches an SSA register and is expanded by querying the points-to analysis, which returns the points-to set $\{$`nil`, `make(chan int, 1)`$^{\top}_{85}$, `make(chan int, 1)`$^{\top}_{92}\}$, where $\top$ denotes an unknown thread. In this case, the analysis aborts because the expanded points-to set contains the channel the fragment was created for. In our experiments we find that channel values are rarely overwritten, so this situation is not common.

The side-effect analysis is a crucial component that tells the abstract interpreter when it can soundly be assumed that channel values are not overwritten in calls to functions outside the fragment. Despite the technical limitations regarding points-to information and potential for analysis failure, the experimental evaluation (Section 4) shows that the analysis is able to find many blocking errors.

3.3 Detecting Blocking Errors

The abstract interpretation of a given program fragment produces a *superlocation graph*, which is a finite transition system where each node denotes a reachable communicating superlocation or is the initial superlocation of the fragment, and edges are obtained from the transitions described in Section 3.2. Given communicating superlocations ϕ_1 and ϕ_2, there is an edge from ϕ_1 to ϕ_2 if the abstract interpreter discovers a sequence of transitions from ϕ_1 to ϕ_2 where all the intermediate superlocations are silent.

The error detection phase is carried out by performing simple model checking on the superlocation graph. Specifically, it checks for every communicating thread g of every superlocation ϕ that there exists at least one path from ϕ to some ϕ' where g has made progress. The absence of such a path indicates a potential blocking error in the fragment.

This approach to detecting blocking errors may have both false positives and false negatives. Since the abstract interpretation phase performs over-approximations, it may discover reachable superlocations that do not correspond to concrete program configurations that are reachable at run-time. Abstract threads in such superlocations may exhibit blocking errors according to the check described above, leading to false positives. Over-approximating whether transitions are enabled can also lead to blocking errors being missed, as the produced transition system may contain spurious paths to superlocations where a thread makes progress.

An error report specifies which thread is potentially blocked and on which line, which channel is involved, and a shortest path in the superlocation graph from the fragment's entry superlocation to the superlocation where the goroutine is blocked. This information aids further diagnosis.

4 EVALUATION

The proposed approach is implemented in the tool Goat, on top of existing libraries for Go program analysis, i.e., the Go package loader, parser, type analysis, SSA IR constructor [23], and the points-to and call graph analysis [22]. We evaluate its efficacy by answering the following research questions:

RQ1: What is the precision of the analysis for detecting blocking errors in real-world Go programs?

RQ2: How does the analysis accuracy compare to that of the state-of-the-art Go concurrency bug detection tools GCatch [14] and GFuzz [13]?

RQ3: Does the analysis scale to large code bases?

RQ4: Are the central design choices (treatment of loops and thread scheduling) important for the analysis accuracy and performance?

To answer the research question regarding accuracy and performance, and to compare with GCatch and GFuzz, we use a suite of 6 large real-world Go projects as benchmarks. These are shown in Table 1. They are mature software systems used in critical production environments. Also, they nearly correspond to the suite of benchmarks used for the evaluation of GFuzz, but we have chosen to exclude Docker as it uses a legacy dependency management and build system. To enable comparison of our results with those of GFuzz, we use the same version of the projects as the GFuzz artifact. We refer to this version of the benchmarks as **suite A**.

The GCatch tool was evaluated on the same projects but in an earlier state, so we manually revert the relevant fixes submitted by the GCatch team such that rediscovering the blocking errors is possible.[8] We refer to this version of the benchmarks as **suite B**.

[8]Reproducible builds are not supported by older versions of Go. Using the same versions of the projects as GCatch was evaluated on is unfortunately not possible as we could not build them in the old state.

When we run Goat on a project, we invoke it once for every package in the project that contains channel creation and consolidate the results over all packages. The pre-analysis phase (Section 3.1) is run once for each package, and the abstract interpretation and blocking error detection phases (Sections 3.2 and 3.3) are run once for each fragment produced by the pre-analysis. We impose a 60 seconds time limit on each run of the abstract interpreter.

The Goat tool and the experimental data are available at https://brics.dk/goat/.

4.1 RQ1: Precision

Analysis precision is measured as the ratio between the analysis' reports that are true positives and the total number of reports.

Running Goat on the 6 real-world Go projects described above results in a set of reports of potential blocking errors. To evaluate the metric the reports must be categorized into true and false positives. Two co-authors manually performed the categorization by inspecting the context of the reported code, and by looking at previously reported and known blocking errors from GCatch and GFuzz, and at the issue trackers for the relevant projects to see if the blocking error had previously been reported or fixed. In cases of doubt due to complex control-flow in the context of the reported blocking error, or insufficient understanding of a project's code, we conservatively categorized the report as a false positive. None of the bugs we found have yet been reported to the developers (except for the *etcd* bug described in Section 2).

The results of the categorization are presented in Tables 2 and 3. In total, Goat reports 99 true positives for suite A (obtained by adding the 'Shared' true positives to those listed in the Goat column in Table 2) and 80 false positives, and 157 true positives and 82 false positives are reported when Goat is run on suite B. Goat achieves a precision of $99/179 \approx 55\%$ on suite A and a precision of $157/239 \approx 65\%$ on suite B, which is on par with the precision of GCatch. This means that roughly one in every two reports is a true positive, which we consider acceptable for practical use.

> *On the suite of 6 large real-world Go projects, Goat achieves an acceptable true positive ratio of more than* 50%.

4.2 RQ2: Comparison with GFuzz and GCatch

We compare the effectiveness of our proof-of-concept blocking error detection tool with two state-of-the-art tools GFuzz and GCatch.

Comparison with GFuzz: For this comparison we run Goat on the benchmarks in suite A and collect the produced blocking error reports. To obtain the blocking errors reported by GFuzz we do not attempt to run the tool itself, as it relies on a highly non-deterministic fuzzing technique. Instead we use the list of blocking error reports in the GFuzz artifact [12]. The results are summarized in Table 2.

The true positive reports are separated into three non-overlapping groups: 'Shared' reports are reported by both tools, whereas the Goat and GFuzz groups denote reports that are produced exclusively by the corresponding tool.

We see that the tools report a nearly disjoint set of blocking errors. This is expected, as the techniques involved are very different. GFuzz only attempts to over-approximate which select branches are

Name	Description	KLOC	GitHub stars
grpc	An implementation of the gRPC remote procedure call system	117	15.5K
etcd	A distributed reliable key-value store	181	39.7K
go-ethereum	An implementation of the Ethereum protocol	368	37.1K
tidb	A distributed HTAP database compatible with MySQL	476	31.2K
prometheus	A monitoring system and time series database	1 186	42.2K
kubernetes	A system for managing containerized applications across multiple hosts	3 453	88.0K

Table 1: Go benchmark projects.

	True positives			False positives
Benchmark	GFuzz	Shared	Goat	Goat
grpc	7	1	2	6
etcd	3	4	31	11
go-ethereum	34	6	14	27
tidb	7	0	0	0
prometheus	8	3	5	6
kubernetes	15	2	31	30
Total	74	16	83	80

Table 2: Bug reports for suite A.

	True positives			False positives	
Benchmark	GCatch	Shared	Goat	Goat	GCatch
grpc	2	4	2	6	1
etcd	2	28	39	11	7
go-ethereum	3	9	25	29	15
tidb	5	0	0	0	3
prometheus	1	7	10	6	2
kubernetes	3	5	28	30	7
Total	16	53	104	82	35

Table 3: Bug reports for suite B.

chosen in an execution, while Goat also explores data-dependent control flow, scheduling, and choice of communication partners.

Since GFuzz detects bugs in fuzzed concrete executions, the tools reports few false positives. Nonetheless, imprecision in how GFuzz tracks which goroutines can send on which channels causes it to report 14 false positives according to its authors. The GFuzz artifact additionally contains bugs reported by GCatch when run on suite A. When the true positive reports are combined with the GFuzz reports, Goat finds 76 bugs that are missed by both GFuzz and GCatch.

Comparison with GCatch: We run both GCatch and Goat on the benchmarks in suite B and collect the produced blocking error reports. The results are summarized in Table 3. The true positive reports are again separated into three non-overlapping groups. The false positive reports are not separated in this fashion.

We find that Goat reports 104 true positives that GCatch misses, whereas Goat misses 16 true positives that GCatch reports. We inspected the 16 reports that Goat misses to understand why they are missed. Three reports involve unsupported features, like reflection, `panic` and `recover`, or standard library functions we have not modeled. Three errors are missed due to imprecision in abstract channel properties, resulting from branches in sequential control flow. GCatch handles such situations by generating different paths for each branching point. Another two errors require precise loop unrolling, handled by GCatch as part of its path unrolling mechanism. One blocking error is missed due to Goat unsoundly pruning the call graph. The remaining errors are undetected due to aborting the analysis, with the unbounded spawning of threads as the predominant factor.

Goat is able to detect many blocking errors that are missed by the state-of-the-art tools GCatch and GFuzz.

4.3 RQ3: Scalability

For this research question we wish to evaluate whether the program analysis scales to large code bases. We do this by measuring the running time of the analysis phases on suite B (the results for suite A are essentially the same). The results of the measurements are presented in Table 4. The "Time" column displays the average time per run, while the "95%" column displays the time t such that 95% of the runs complete in less than t seconds.

Across all benchmarks, pre-analysis is performed 476 times, whereas the abstract interpretation phase and the blocking error detection phase are run 11 199 times. On average, 14 minutes are spent in each run of the pre-analysis phase and a second is spent in each run of the abstract interpretation phase. In these experiments we repeat the pre-analysis for all program packages to best measure the potential of our approach in detecting blocking errors. It is also possible to run the pre-analysis once for all packages combined, at the cost of a modest reduction in analysis precision. The largest benchmark, *kubernetes*, is an outlier for the pre-analysis time but is processed quickly by the main analysis phases. *tidb* is an outlier for analysis time where more than 5% of analysis runs for this benchmark time out.

A detailed breakdown of the running time of the abstract interpretation phase is presented in Figure 6. Here we see that the vast majority of runs of the abstract interpreter finish within 5 seconds, and that only a small number of runs (70 of 11 199 $\approx$ 0.6%) are aborted due to reaching the time limit. The time required for the blocking error detection phase is negligible.

The abstract interpretation and blocking error detection phases of the approach finish quickly in the majority of cases. The total analysis time is dominated by the points-to analysis performed in the pre-analysis phase.

	Pre-analysis			Main analysis		
Benchmark	Runs	Time	95%	Runs	Time	95%
grpc	96	24 s	46 s	3 023	1.64 s	5 s
etcd	45	42 s	65 s	1 769	0.12 s	1 s
go-ethereum	65	31 s	54 s	3 796	0.55 s	2 s
tidb	14	1 182 s	1 958 s	342	10.99 s	60 s
prometheus	24	67 s	494 s	742	0.22 s	1 s
kubernetes	232	1 651 s	8 924 s	1 527	0.15 s	1 s
Total	476	856 s	7 774 s	11 199	1.02 s	4 s

Table 4: Number of runs and running times for pre-analysis and main analysis.

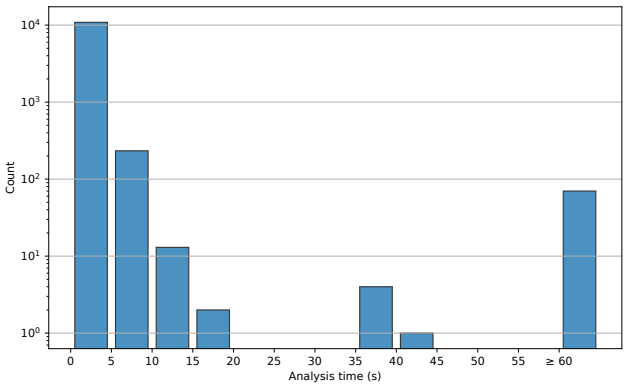

Figure 6: Distribution of running times.

Mode	TP	FP	Aborts	Timeouts
Normal	123	53	76%	69
Sound loops	109	46	79%	66
Fine-grained scheduler	113	47	73%	780

Table 5: Importance of design choices.

4.4 RQ4: Importance of Design Choices

For this research question we wish to evaluate the choice of modeling only one iteration of each loop and the coarse-grained thread scheduler, both explained in Section 3.2. We do this by analyzing the benchmarks in suite B with Goat in three different modes. 'Normal' is the default mode, 'Sound loops' models loops soundly as in normal abstract interpretation, and 'Fine-grained scheduler' replaces the coarse-grained scheduler with a fine-grained scheduler that considers the interleavings of all operations. In Table 5, we compare each mode by measuring reported true and false positives, timeouts, and the abort ratio for all benchmarks except *kubernetes* (which we exclude in this experiment due to prohibitive pre-analysis times).

Disabling the special treatment of loops increases the number of aborted analysis runs, as expected. The analysis also detects fewer blocking errors: 16 errors are missed compared to a normal run, but only 2 new errors are discovered. The number of false positives also decreases, but at a smaller rate compared to that of true positives.

Modeling all interleavings, including those for silent superlocations, causes a significant increase in the size of the reachable superlocation space explored by the worklist algorithm. For example, analyzing the fragment in Figure 1 with the approach from Section 3.2 only computes 637 unique superlocations, while 29 629 superlocations are reached if considering all possible interleavings. More generally, the average time for one abstract interpretation run increases from 1.02 s to 9.43 s, the number of timeouts increases substantially, and the number of detected errors decreases without any significant reduction in false positives.

The pragmatic design choices positively impact the analysis by improving the error detection capability and efficiency.

4.5 Discussion

Among the threats to the validity of the claims is the choice of benchmarks, which may not be representative of typical Go programs. We have selected benchmarks that are used in prior work to enable comparison, and they are large and high ranked on GitHub. Also, mistakes during the manual inspection of reports may lead to incorrect classification of true positives and false positives; as mentioned we have attempted to conservatively classify difficult cases as false positives.

In our experiments the abstract interpretation phase aborts in 73% of all runs according to the conditions described in Section 3.2 and reports no blocking errors in those cases. Although the tool still finds many blocking errors and has a good ratio between true and false positives according to the experiments, many errors may remain undetected. As an interesting opportunity for future work, it may be possible to increase the completion rate by refining the analysis domain and the heuristics of the pre-analysis.

The GoBench [26] test suite contains 103 *bug kernels*, i.e., Go programs that have been manually synthesized by extracting the critical parts of surrounding code necessary to trigger blocking errors found in real-world Go projects. Of these 103 kernels, 24 are synthesized from blocking errors involving channels and are therefore in the scope of Goat. We find that Goat detects 12 of the 24 blocking errors (whereas GCatch detects 10), again indicating that there may be many more errors to be found (provided that the bug kernels are representative of real-world Go programs). Alongside over-approximations and aborting the analysis, another prevalent cause of missing blocking errors is the construction of $\mathcal{P}$ and $\mathcal{F}$, which both Goat and GCatch rely on. The current heuristics largely group channels intra-procedurally, but several bug kernels expose blocking errors caused by intricate inter-procedural interactions. More extensions for identifying specific inter-procedural patterns (such as C3 in Section 3.1) might improve error detection capabilities while keeping the resulting fragments small, which remains to be explored in future work.

Many false positives are due to channel operations that are unreachable in concrete executions but reached by the analysis due to over-approximation. Typical causes of such over-approximations are data-dependent control-flow, spurious cycles in the call graph, imprecision due to channel inclusion in dynamic data structures, or excluding critical channels from the relevant set of a given fragment. Under-approximations also cause false positives. Call graph pruning at highly imprecise call sites may remove edges to functions

that should be included in the fragment, and the special treatment of loops and the coarse-grained modeling of the thread scheduler might lead to missing control flow paths. As pointed out in Section 3.3, over-approximation in the pre-analysis and abstract interpretation phases may also cause errors to be missed in the blocking error detection phase.

The results of the experiments for RQ3 indicate that Goat would benefit from more performant analyses used in the pre-analysis phase. Our approach does not require the full points-to solution that the analysis produces, it only needs points-to information for channel operations and for reference values that come from outside the fragment being analyzed. This suggests that a demand-driven points-to analysis would be a good fit for Goat.

To limit the scope of Goat, we focus on blocking errors, but the underlying technique may be extended to check for other channel-related properties. The produced abstract states may be used to check for potential safety violations, such as writing to or closing already closed or `nil`-valued channels. The scope of blocking errors can also be extended to include other common Go concurrency primitives, like `Mutex`, `WaitGroup`, and `Cond`.

5 RELATED WORK

As stated in Section 1, GCatch [14] and GFuzz [13] represent the current state-of-the-art for automated detection of blocking errors in real-world Go programs. The key difference to our approach is that GCatch relies on SMT encoding of the execution paths in the program fragments whereas Goat builds abstract state spaces for the program fragments using abstract interpretation. We believe abstract interpretation provides a natural approach to model the various language features of Go, and that it is flexible for exploring variations of the abstract domain. GFuzz is a dynamic analysis tool that instruments `select` statements to force execution of specific branches and produce alternative case selections via fuzzing. This leads the modified executions to reach execution paths that may be difficult to produce in traditional testing. It also instruments the run-time of Go by collecting relationships between threads and channels, and periodically scans the memory for threads blocked on channels with no future communication partner. As a dynamic analysis tool, GFuzz has high precision for the paths it explores but is restricted to concrete executions.

Combining behavioral types and model checking is another popular approach. Techniques include deadlock detection for programs with synchronous channels by global graph synthesis [19] and session type inference for fenced programs [10], abstracted to a symbolic state space for which safety and k-liveness properties are verified. The Godel checker [11] infers session types and verifies safety and liveness properties defined as μ-calculus encodings via off-the-shelf LTL model checkers. Key limitations to these approaches are difficulties in combining session types and more precise data abstractions, and scalability to real-world Go programs due to lack of coverage of other Go features, e.g., higher-order functions, aliasing, and interfaces. The most recent approach in this family is Gomela [5, 6], which translates Go programs to Promela and uses SPIN [7] for model checking. The experimental results reported for Gomela show good results on small programs but also that it finds relatively few concurrency errors in large, real-world Go programs.

Abstract interpretation has a solid mathematical foundation [3] and has been studied and applied extensively for decades, mostly for single-threaded programs but also for concurrency (see, e.g., [9, 16–18, 25]). Most of the existing techniques are based on thread-modular analysis, without localization to program fragments, and are designed for shared-memory concurrency not involving channels. Previous work on abstract interpretation for channel-based concurrency introduced a notion of process-local static analysis where each abstract thread flow-sensitively models an over-approximation of possible futures as lattice-valued regular expressions [15]. However, it only models communication for a fixed number of threads and synchronous channels, and has only been evaluated on small programs, unlike our approach.

Many of these existing tools detect not only blocking errors but also other kinds of concurrency errors. In principle, Goat can easily be extended to also scan for safety errors, based on the superlocation graphs it already produces, but we leave that for future work.

GCatch, Gomela and Goat all achieve scalability by analyzing program fragments individually. The technique Goat uses for approximating the behavior of program code outside the fragment being analyzed can be seen as a variation of the "worst-case separate analysis" approach by Cousot and Cousot [4], except that we do not need to compose modular analysis results. Also, we leverage the pre-analysis and we choose to ignore certain potential side-effects involving references as discussed in Section 3.2.

6 CONCLUSION

We have shown that localized abstract interpretation is a promising approach to detect blocking errors in programs that use channel-based thread communication. This approach offers an alternative to existing techniques that rely on SMT encoding or bounded model checking of program fragments. The pragmatic design choices make the approach neither sound nor complete, but enable scalability to large code bases and detection of many bugs in practice. The implementation of the approach, Goat, can detect blocking errors in real-world Go programs that other tools miss, with more than 50% of the reported issues being true positives.

Our experiments also suggest opportunities for further improvements. Provided that the existing collection of small benchmark programs by Yuan et al. [26] is representative of real-world usage of Go, many blocking errors remain beyond reach of existing automated techniques despite the progress obtained by Goat. It may also be worthwhile to develop more specialized pre-analyses. Furthermore, it may be interesting to extend the analysis to also report safety errors and to model other concurrency primitives, by building on the superlocation graphs produced by Goat and making further use of the flexibility of abstract interpretation.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. Dissertation. University of Copenhagen.
[2] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, PLDI 1990.* ACM, 296–310.
[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation

of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. ACM, 238–252. https://doi.org/10.1145/512950.512973

[4] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*. Springer, 159–178. https://doi.org/10.1007/3-540-45937-5_13

[5] Nicolas Dilley and Julien Lange. 2020. Bounded Verification of Message-Passing Concurrency in Go using Promela and Spin. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020 (EPTCS, Vol. 314)*. 34–45. https://doi.org/10.4204/EPTCS.314.4

[6] Nicolas Dilley and Julien Lange. 2021. Automated Verification of Go Programs via Bounded Model Checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1016–1027. https://doi.org/10.1109/ASE51524.2021.9678571

[7] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[8] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7 (1977), 305–317. Springer.

[9] Markus Kusano and Chao Wang. 2016. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, 799–809. https://doi.org/10.1145/2950290.2950291

[10] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing Off Go: Liveness and Safety for Channel-Based Programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 748–761. https://doi.org/10.1145/3009837.3009847

[11] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 1137–1148. https://doi.org/10.1145/3180155.3180157

[12] Ziheng Liu, Yu Liang, Shihao Xia, Linhai Song, and Hong Hu. 2021. *GFuzz ASPLOS 2022 #710 Artifact*. https://doi.org/10.5281/zenodo.5893373

[13] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. 2022. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 888–902. https://doi.org/10.1145/3503222.3507753

[14] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 616–629. https://doi.org/10.1145/3445814.3446756

[15] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. 2018. Process-Local Static Analysis of Synchronous Processes. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002)*. Springer, 284–305. https://doi.org/10.1007/978-3-319-99725-4_18

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

[17] Antoine Miné. 2014. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*. Springer, 39–58. https://doi.org/10.1007/978-3-642-54013-4_3

[18] Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. 2016. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. Toulouse, France. https://hal.archives-ouvertes.fr/hal-01271552

[19] Nicholas Ng and Nobuko Yoshida. 2016. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. ACM, 174–184. https://doi.org/10.1145/2892208.2892232

[20] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 17–30. https://doi.org/10.1145/1926385.1926390

[21] The Go Authors. 2010. Share Memory by Communicating. https://go.dev/doc/effective_go.

[22] The Go Authors. 2022. Points-To analysis and Call Graph construction for Go. https://pkg.go.dev/golang.org/x/tools@v0.1.10/go/pointer

[23] The Go Authors. 2022. Static Single Assignment Intermediate Representation for Go. https://pkg.go.dev/golang.org/x/tools@v0.1.10/go/ssa

[24] 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, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 865–878. https://doi.org/10.1145/3297858.3304069

[25] 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, ASE 2016, Singapore, September 3-7, 2016*. ACM, 391–402. https://doi.org/10.1145/2970276.2970337

[26] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 187–199. https://doi.org/10.1109/CGO51591.2021.9370317