GoSonar: Detecting Logical Vulnerabilities in Memory Safe Language Using Inductive Constraint Reasoning

Md Sakib Anwar
The Ohio State University
anwar.40@osu.edu

Carter Yagemann
The Ohio State University
yagemann.1@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

***Abstract*—As the global community advocates for the adoption of memory-safe programming languages, a significant research gap persists in identifying the critical vulnerabilities that follow. Logical vulnerabilities represent the most formidable threat to these programs, in the absence of memory safety related vulnerabilities such as buffer overflow. Go, a prevalent memory-safe language for cloud-based applications where resource availability is paramount, is especially susceptible to nonterminating, resource-exhaustive vulnerabilities. We present a novel approach to the problem, *inductive constraint reasoning*, designed to evaluate nontermination in complex, real-world programs, demonstrating superior performance compared to contemporary tools on a standardized dataset. Our methodology employs binary-level underconstrained symbolic execution to gather the constraints necessary for multiple recursive iterations. By applying a first-order derivative to these constraints, we model and classify various recursive functions, determining whether their subgoals converge to a global objective. This study addresses numerous challenges in the analysis of Go programs while simultaneously developing and implementing a practical solution to detect uncontrolled recursion, which has revealed 5 new vulnerabilities in the Go standard library.**

1. Introduction

In a recent report [2] issued by the US White House Office of National Cyber Directory (ONCD), the integration of memory-safe programming languages has been identified as a key element in the formulation of the National Cybersecurity Strategy. This strategic decision is underpinned by the revelation that 70% [3] of all Common Vulnerabilities and Exposures (CVEs) are attributable to memory safety issues. Despite concerted efforts of both industrial entities and academic institutions to develop frameworks [1], [4], [18] that aid in the detection and mitigation of such vulnerabilities, eradication of these issues cannot be achieved without adopting such languages [2]. Furthermore, traditional justifications for selecting conventional programming languages, such as execution speed [14], [15], size of compiled binaries [23], and the ability to compile into native executables [24]—are increasingly being challenged by modern memory-safe languages like Go and Rust given the growing availability of resources.

Nevertheless, there is a noticeable lack of research focused on program analysis to detect vulnerabilities in memory-safe languages. An in-depth review of Go related CVEs shows that logical vulnerabilities are the most common. Considering Go's extensive use in the backend of cloud-based platforms [10] such as Uber, DropBox, Meta, and Netflix, logical vulnerabilities that can impact availability represent a major risk. OWASP ranks unrestricted resource consumption 4^{th} among the top 10 threats to API security in 2023 [16]. Interestingly, among the leading CWEs for Go are Uncontrolled Resource Consumption (CWE-400), Allocation of Resources Without Limit (CWE-770) and Uncontrolled Recursion (CWE-674), which represent 79 (35%) of the CVEs. Specifically, logical vulnerabilities that can be exploited to deplete resources can lead to financial losses through denial-of-service attacks. Furthermore, we identified six instances of uncontrolled recursion in Go's *standard library*, with five of them having a Common Vulnerability Scoring System (CVSS) rating of 7.5 in 2022 alone. These vulnerabilities in the standard library make any Go-based application vulnerable to resource exhaustion.

Unlike traditional DoS/DDoS, logical vulnerabilities weaponized to cause denial of service via resource exhaustion do not require multiple requests and thus can avoid detection by traditional methods. The difficulty in verifying these vulnerabilities adds to the complexity, especially when dealing with uncontrolled recursion. Identifying the start of an uncontrolled recursion and demonstrating the lack of a clear termination condition, particularly when it spans multiple functions, are some of the challenges faced. Moreover, unlike traditional languages, Go incorporates various compilation designs that make analysis of Go programs using traditional program analysis difficult. Although loops, including infinite loops, present similar challenges, they do not engage additional computational or memory resources, thereby rendering them less susceptible to exploitation compared to recursive constructs.

Existing Approaches. A considerable body of research has been devoted to the detection of nontermination in relatively straightforward programs. These programs are typically integer-based and resemble C-like syntax. Researchers have utilized methodologies such as program analysis, theorem resolution, and the construction of test oracles for

runtime assessment. Regrettably, recent investigation [21] has demonstrated these methodologies to be ineffectual beyond their own dataset, when applied to real-world applications, even those developed in lower-level languages such as C/C++. Furthermore, it also identifies pointer manipulation, array, data structure and recursion as key failure points for these tools [21]. Interestingly, programs built with contemporary memory safe languages employ all of these heavily in order to be efficient and secure which makes these tools inadequate for analysis.

Nontermination work divides into termination and nontermination detection. Proteus [25] summarizes loops, while UAutomizer [12] translates traces into automata for termination analysis. A data-driven approach [28] uses ML to learn loop limits. Nontermination detection, such as TNT [11], generates counterexamples from recurrent states. Another work [7] reverses transition systems in nondeterministic integer programs. Loopster [26] and its continuation [27] reduces nontermination to a reachability problem and use path dependency automata. Practical solutions like 2LS [19] use interprocedural abstract interpretation for C programs, and AProVE [8] employs constraint solving for recurrent sets in loops. One work [29] addresses infinite loops in real-world programs, finding C/C++ vulnerabilities with 240 hours of fuzzing. However, none efficiently handle complex real-world programs, let alone Go programs.

Our Proposal. Drawing from initial observations and the gap in existing approaches, we introduce an innovative, efficient and practical framework designed to identify uncontrolled recursion in real-world Go programs, GoSonar. Just as sonar technology is used to measure the depth of the sea, GoSonar can detect the depth of recursions detecting uncontrolled recursion vulnerabilities. Our method utilizes binary-level underconstrained symbolic execution to collect constraints necessary for several recursive iterations. We employ a first order derivative of these constraints to model and classify different kinds of recursive functions. Specifically, we analyze the change of constraints from $n-1^{th}$ recursion to n^{th} recursion i.e. the new set of constraints each recursion adds. This analysis helps identify the trajectory of program execution along with the recursion's subgoal and determines whether this subgoal converges to a global goal or not. A termination condition will be part of this delta and can not be met indefinitely leading the recursion closer to a termination state.

Unlike previous studies that applied constraint solving to generate a repetitive set of program states leading to infinite recursion, our approach adopts an inductive strategy focusing on constraint changes. Although inductive reasoning has been used to simplify constraint solving following inductive logic programming, we use it to predict the trajectory of program execution. This approach allows for the assessment of complex recursions in real-world programs, where the route to a recursion may be intricate, yet the recursion itself can be straightforward. By eliminating the need to analyze all the constraints and focusing on their differences, our approach effectively narrows the scope of the decision-making process. We address several challenges while designing our prototype, GoSonar, related to symbolic execution of binaries compiled from a heavily instrumented language, enriching the process of detection of potential candidates and finally systematic exploration of different kinds of recursive function and their effect on constraints.

Evaluation & Result. We used GoSonar to analyze all 36 packages in the core Go standard library used by every Go program in some form. GoSonar identifies 14 packages with reachable recursions and examines these packages to detect 983 recursions, forming 20,036 lassos composed of the recursion and a stem. GoSonar uncovered 5 new vulnerabilities in the Go standard library, all reported and acknowledged by developers[1]. In order to test the soundness of GoSonar, we have evaluated GoSonar on a dataset [21] of infinite recursion vulnerabilities in OSS programs. This dataset contains both vulnerable (nonterminating) and patched (terminating) versions of each bug and provides a taxonomy of such vulnerabilities based on the root cause. Furthermore, due to the lack of tools that can work with Go programs, we compared GoSonar against leading technologies for native language, namely UAutomizer [12], CPAchecker [5], 2LS [19], AProVE [8], and T2 [6] on the same data set. These applications set the standard in termination analysis, with GoSonar exceeding their performance.

Contribution. The contribution of this work can be summarized as the following:

- We present GoSonar, the first and only tool of its kind that can find real uncontrolled recursion vulnerabilities in Go programs.
- We analyze all the standard libraries and find 5 new uncontrolled recursion vulnerabilities.
- We propose a novel approach, *inductive constraint reasoning* for evaluating nontermination in complex real world programs that outperforms all contemporary tools on a standardized dataset.

2. Overview

This section provides a formal definition of recursions and their essential role in contemporary programming. Then illustrates an example with an incomplete patch to emphasize the importance of automated detection and the limitation of manual analysis. Lastly, it addresses the difficulties in developing a detection tool and the presumptions considered during the design of GoSonar.

2.1. Background

A recursion in a program is typically modeled as a *lasso* consisting of a *stem* and the *recursion*. The stem provides a path connecting the recursion back to the entry point of the

1. We have followed Go's disclosure guideline for reporting vulnerability and got acknowledgment from Go developers

```
1 func (dec *Decoder) decIgnoreOpFor(wireId typeId,
    inProgress map[typeId]*decOp) *decOp {
1 func (dec *Decoder) decIgnoreOpFor(wireId typeId,
    inProgress map[typeId]*decOp, depth int) *decOp {
2     if depth > maxIgnoreNestingDepth {
3         error_(errors.New("invalid nesting depth"))
4     }
      ...
5     wire := dec.wireType[wireId]
6     switch {
7     case wire == nil:
8       errorf("bad data: undefined type %s",
            wireId.string())
9     case wire.ArrayT != nil:
10      elemId := wire.ArrayT.Elem
11      elemOp := dec.decIgnoreOpFor(elemId,
                    inProgress)
11      elemOp := dec.decIgnoreOpFor(elemId,
                    inProgress, depth+1)
      ...
12    case wire.StructT != nil:
13      enginePtr, err :=
          dec.getIgnoreEnginePtr(wireId)
      ...
14 }
```

Figure 1: [encoding/gob/decode.go]
Patch for uncontrolled recursion in CVE-2022-30635

program, creating a complete path to invoke the recursion. Recursion serves as a fundamental component within the domain of functional programming, particularly in scenarios where a given problem can be decomposed into analogous, smaller sub-problems, progressively advancing towards a comprehensive solution through their resolution. Parsing nested data structures is one of the primary applications of recursion, especially when dealing with nested input data. However, while it is theoretically feasible to transform a recursive algorithm into an iterative one, such a conversion requires the introduction of an auxiliary function to call the target function repeatedly. The mere inclusion of a depth parameter for the termination condition evaluation often proves inadequate for larger-scale projects, as we will illustrate in our motivating example. Specifically, the example will highlight a prevalent instance of recursion within the Go standard library, focusing on the inadvertent occurrence of uncontrolled recursion and highlighting the deficiencies in how developers currently attempt to patch such issues.

2.2. Motivating Example

CVE-2022-30635 is an uncontrolled recursion in the Go `encoding/gob` package from versions 1.17.12 to 1.18.4. Package gob handles binary values (gobs) exchanged between an Encoder and a Decoder, commonly used for transporting RPC arguments and results. An attacker could trigger a program panic via stack exhaustion with deeply nested structures. Reported in June 2022 and marked resolved a month later, GoSonar finds that the vulnerability persists in the latest Go version due to an incomplete patch which was later assigned CVE-2024-34156 [9]. The issue is in `decIgnoreOpFor`, as shown in Figure 1 with red for vulnerable and green for patched version. The function calls itself on line 11 if `wire` is `Array`. For nested arrays, `elemId` in line 10 will be the same as `wireId` at line 1, preventing termination. The patch adds a `depth` argument, incremented at the recursive call on line 11, with a check at line 2 to terminate the recursion if `depth` exceeds `maxIgnoreNestingDepth`.

Unfortunately, while this patch fixes one path that triggers the bug, it misses another larger recursion that is still uncontrolled. As shown in Figure 1, a call to the function `getIgnoreEnginePtr` is present in line 13 when the variable `wire` is a structure. This function in turn calls the function `compileIgnoreSingle`, which subsequently calls the function `decIgnoreOpFor`, creating a recursive loop. Figure 2 illustrates this path using a red arrow. The patch does not properly control this larger recursion because the `depth` variable on which it is based is reset to 0 each time the program enters the function `getIgnoreEnginePtr`, undermining the check added at line 2. Specifically, the patch addresses the recursion for a nested array; however, it misses a larger recursion occurring for a nested data structure. This motivates the need for a novel solution that can allow developers to systematically find these trickier uncontrolled recursion vulnerabilities. Manual analysis and existing techniques are insufficient.

2.3. Challenges

There are several obstacles in uncovering recursions and verifying whether there is a path that can make them uncontrolled using contemporary tools. Additionally, analyzing binaries for compiled memory-safe languages such as Go introduces further obstacles due to their esoteric design and compiler optimizations. We had to overcome all of these challenges to build GoSonar:

Recursion Detection. The first step in detecting uncontrolled recursions is to identify all recursions in the program. Although this is straightforward for older languages like C/C++, it is more complex for modern languages due to the use of advanced data structures, instrumentation, and interfaces. For example, consider the function call on line 11 from the motivating example (Figure 1). The function is a method for `Decoder`, as seen in the prototype on line 1. At the binary level, such calls become indirect calls. Debug symbols help, but not for interface calls resolved at runtime. The same applies to calls on member objects initialized during object creation. Moreover, Go's interfaces can be anything, even without functions (`interface {}`), and are not explicitly declared like in languages with `implements`. These nuances complicate finding recursions or function calls.

Reachability Detection. Reachability determination is crucial for developers who overlook non-exploitable bugs. Uncontrolled recursion in a private function (not accessible to outside actors) may have termination enforced by a stem starting from a publicly accessible point. Most functions require data structure initialization before calling the target function. Accurate analysis needs proper initialization,

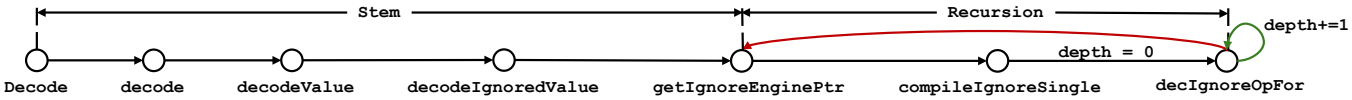

Figure 2: Lasso in motivating example highlighting the larger missed recursion in patch for CVE-2022-30635

knowledge of the code, and use cases, which current techniques cannot automate. For instance, in Figure 1, the recursive function `decIgnoreOpFor` of structure `Decoder` is private. We find that the public function `Decode` initializes the data structure. Finding reachability paths in Go is hard due to frequent indirect calls. The main approach, dynamic exploration or fuzzing, is time-consuming and doesn't scale. Fuzzing frameworks cannot generate the necessary objects to trigger logical vulnerabilities. Lastly, even if backtracing is possible, deciding when to stop is tough, as invoked functions might be private and unreachable by an attacker.

Proving Uncontrolled Recursion. Once a recursion is discovered and verified to be reachable, the challenge is to determine whether it can enter an uncontrolled state that constitutes a vulnerability. For example, a recursion may loop 10,000 times but still manage to always terminate, or there may exist a path that consumes so many resources that only 100 iterations are needed to exceed the machine's limits. In either case, naive verification based solely on the depth of the recursion is ineffective and imprecise. Contemporary tools have tried to subvert this by incorporating some form of dynamic analysis to generate cases or by converting the lasso into automaton by formally proving the termination. However, both approaches prove unrealistic for a modern program where the termination condition can be extrapolated from pointer resolution, function calls, and instructions that use complicated nested data structures. In contrast, pure static analysis on a function level may detect `depth > maxIgnoreNestingDepth` as a terminating condition in our motivating example; however, it may not consider all possible paths to reach the function or overapproximate possible values for the various variables.

Analyzing Go Executable. Lastly, unlike other languages, Go lacks a static ABI or call convention, instead opting to use an optimized convention that changes per function. The passing of arguments varies according to the prototype. For example, some are passed via the stack, and others are passed using registers. The ordering is not fixed like in C/C++ programs. For data structures, complexity dictates whether fields are flattened into primitive types. Moreover, as a memory-safe language, Go inserts extensive checking routines for memory management and garbage collection during compilation, complicating binary analysis. Consequently, resource-intensive analysis techniques, such as symbolic execution, face an increased risk of path and state explosion. These factors hinder the application of typical program analysis techniques to Go programs.

2.4. Scope

In addition to the technical challenges described previously, we also have the goal of designing GoSonar to be useful to Go developers and security analysts. The Go community already has a robust method for tracking security issues in which any reported vulnerability is quickly assessed, tested, fixed, and made public with an appropriate CVE advisory. Moreover, Go already has several static analysis tools, such as GoSec [20], GoKart [17], and Staticcheck [13], which use source code level checking to detect common vulnerabilities. However, they fail to detect the vulnerabilities targeted by GoSonar due to their lack of applicability and context. Furthermore, like most languages used for development, Go has a large repository of third-party packages that make development efficient and streamlined. The current system for enrolling a new package includes a step that vets for vulnerabilities. Our vision with GoSonar is that it can be incorporated into the pipeline for publishing new packages to vet logical vulnerabilities that are currently undetected. Although we focus on resource exhaustion that results from uncontrolled recursion in this work, we expect that in the future, GoSonar can be extended to model a wide range of logical vulnerability classes.

2.5. Assumptions

GoSonar is a binary analysis tool and can detect and prove uncontrolled recursions with symbolic execution in Go executables. We assume that the binary contains debug symbols to facilitate binary analysis. Moreover, since we expect the user of GoSonar, such as the package vetting pipeline described previously, to have access to the target source code, we allow an optional component of our analysis to take advantage of the source code. We further discuss the effect of source code in our analysis in our evaluation and ablation study. We assume that the program can be compiled into a native executable using the flag `--static-libgo`, resulting in a monolithic standalone executable that is easier to analyze.

For our threat model, we assume that the attacker has the ability to interact with the target program via the public functions of the packages, and no additional check is done to sanitize the input to these functions other than what is already in place. We also assume that the attacker can use any other packages to facilitate the creation of the payload. For example, the payload may be crafted using packages such as `reflect` and `unsafe` if desired. Given the positive response of Go developers during the disclosure of the findings of GoSonar, our threat model appears reasonable.

Lastly, we define a resource exhaustion vulnerability as one that can continue increasing the usage of resources

without any warning or error condition. We define an uncontrolled recursion as one that can continue to loop indefinitely until the program crashes or gets stuck.

3. Design

This section explores our understanding and its role in addressing the challenges in §2.3. Our key insight for successful under-constrained symbolic execution is that public functions in a Go program are easily identifiable and often initialize subsequent private functions containing recursion. However, symbolic execution sometimes needs guidance and help in analyzing constraints for verifying uncontrolled recursion.

GoSonar consists of four key components: the first component, as explained in §3.1, identifies all potential candidates for uncontrolled recursion within a Go program. Once these candidates are identified, their reachability is validated by the second component (§3.2). Subsequently, the reachable recursions or lassos are symbolically executed to gather constraints using the third component (§3.3). Lastly, these constraints are employed to demonstrate that the recursions are uncontrolled, as discussed in the fourth component (§3.4). Below, we provide an overview of these four main components of GoSonar:

3.1. Candidate Finder

GoSonar generates all possible candidates for uncontrolled recursion by identifying all recursions not only in the program but also in the library. As mentioned earlier scaling to the level of a modern program with thousands of library functions is a difficult task. GoSonar approaches this problem with a modularized solution, where we realize that state-of-the-art graph algorithms can be used to find *circuits* in the Control Flow Graph (CFG). However, loops or circuits in the CFG may denote both loops and recursions. In order to find only recursion, GoSonar needs to evaluate the Callgraph instead of the CFG.

Callgraph Generation. GoSonar uses static analysis to analyze the executable binary and build a callgraph for the program. To transform the CFG to a call graph, all the nodes representing basic blocks of a function can be merged together, making one node for each function. Furthermore, only the edges generated from a `CALL` in the binary can be used to transform the CFG to a much smaller callgraph. However, we find even after this downsizing, the graph remains considerably big with thousands of functions and edges. Upon investigation, we find many of these functions are stub functions which are generated during compilation as a placeholder. Furthermore, the runtime library, which is responsible for managing the runtime environment such as memory management and thread management, adds a lot of function calls. After pruning the graph with these rules GoSonar can reduce the size enough for efficient circuit detection in the graph.

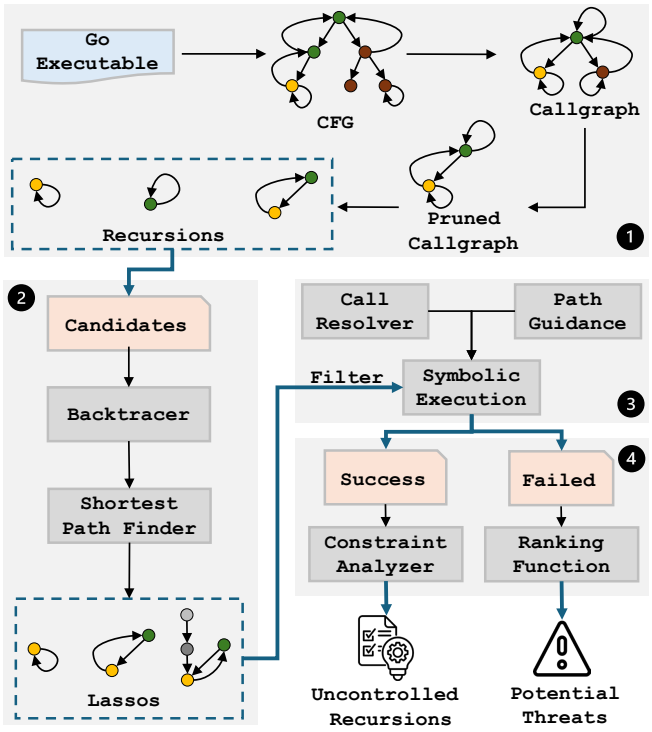

Figure 3: Overall design of GoSonar
❶ Candidate Finder (§3.1) ❷ Reachability Verifier (§3.2) ❸ Symbolic Execution Guide (§3.3) ❹ Constraint Reasoning (§3.4)

The entire process can be seen in Figure 3 – ❶ where the native Go executable is used to generate a CFG that can contain multiple nodes for each function represented as different colored nodes. Upon merging these nodes and edges, GoSonar generates the callgraph, which is further pruned to only keep functions of interest. The generated recursions contain both self-loops and recursions spanning multiple functions.

3.2. Reachability Verifier

After detecting all recursions, their reachability and exploitability by an attacker must be verified. In Go, naming conventions determine the scope of functions or data structures instead of keywords such as `private` or `public`. Names starting with a capital letter are public, while others are private. Public function-induced recursions can be easily triggered by attackers. For recursions involving only private functions, a public function can still make them reachable. An attacker can build an input to a public function to reach a targeted private function. GoSonar verifies the reachability of candidates in two steps:

Backtracing for Public Functions. The callgraph can be used to find all callers for a given node, however, given the complexity of symbolic execution if the stem is too large it might cause a problem in finishing the symbolic execution. Thus GoSonar runs a depth first backtracing for each

```
1  func Unmarshal(data []byte, v interface{}) error {
2    return NewDecoder(bytes.NewReader(data)).Decode(v)
3  }
4
5  func (d *Decoder) Decode(v interface{}) error {
6    return d.DecodeElement(v, nil)
7  }
8
9  func (d *Decoder) DecodeElement(v interface{},
             start *StartElement) error {
10   val := reflect.ValueOf(v)
11   ...
12   return d.unmarshal(val.Elem(), start)
13 }
14
15 func (d *Decoder) unmarshal(val reflect.Value,
             start *StartElement) error {
16   ...
17   switch v := val; v.Kind() {
18     ...
19     case reflect.Slice:
20       if err := d.unmarshal(v.Index(n), start);
21       ...
22   }
23 }
```

Figure 4: [encoding/xml/read.go]
Public functions leading to victim private function `unmarshal` for CVE-2022-30633

function in the candidate recursion. This helps GoSonar limit the length of the stem. While backtracing whenever a public function is found, it is added as a potential entry point for the underconstrained symbolic execution. However, we notice that, although counter intuitive, longer stems sometimes produce better results given the initialization undertaken by each function. Thus, GoSonar does not stop upon encountering a public function; rather, the stopping condition is entirely based on the predefined depth limit that can be configured easily.

Consider CVE-2022-30633 an uncontrolled recursion in the encoding/xml package that spans multiple private functions including `unmarshal`. The recursion is uncontrolled when the input variable `val` is a `slice` as can be seen on line 20 of Figure 4. There are multiple public functions leading to the function `unmarshal` as shown in Figure 4, namely `Decode`, `DecodeElement` and `Unmarshal`. The closest public function to the target function, `unmarshal` is `DecodeElement` which calls on line 12. However, it can be noticed that this function also provides the least amount of context for the target function, leading to more constraint solving for symbolic execution. For example, this function does not initiate the value of the variable `start` whose value in turn decides whether the function `Token` will be called on line 18 or not. Whereas the function `Decode` sets the value of `start` to `nil` by default on line 6. Similarly, the structure `Decoder` uses a `reader` interface in many parts of the function `unmarshal`. Many structures implement this interface; however, the function `Unmarshal` initializes the reader to be `byteReader` on line 2, thus reducing the execution tree by concretizing the reader type.

Shortest Path Finder. The order in which all the paths from a public to a private function is discovered in a depth first search is uncertain. Thus, to confirm that the shortest path from a public function is analyzed, GoSonar uses the callgraph to find the shortest path between the two functions. That is, from our example for the recursion in `unmarshal`, GoSonar needs to find the shortest path between `Unmarshal` and `unmarshal`. This is an optional and configurable optimization step that helps the symbolic execution to be more efficient. All of this process can be seen in Figure 3 – ❷ where the candidates are passed to the backtracer, which in turn uses the shortest path finder component to find all possible lassos. We see that not all recursions from the last step end up generating lassos, whereas depending on each function of a recursion there can be multiple lassos with varying length of stem. This ensures that GoSonar covers all possible ways to reach a recursion and consider the constraints necessary that might lead to a recursion becoming uncontrolled.

3.3. Symbolic Execution Guide

The goal of symbolic execution is to gather the constraints while executing the program from a publicly accessible function to the recursion and subsequently executing the recursion multiple times. All the inputs to the entry point of the stem (i.e. the public function) that an attacker can interact with are unconstrained for an underconstrained symbolic execution. However, a symbolic execution will evidently face resource exhaustion while trying to detect other potential cases. To this end, all the lassos created in the last step go through a filter that decides eligibility based on the number of functions in the recursion and the stem. This filter is configurable and can be changed according to the need for our purpose.

However, the filter alone is not enough and thus the execution needs to be guided to avoid state and memory explosion. That is, upon each step, a priority has to be made for all the successor states such that the execution can find the recursion efficiently and comprehensively. Moreover, the symbolic execution can get stuck if it is unable to resolve indirect function calls such as interface function calls, which are resolved at runtime. GoSonar employs two tactics to address these challenges:

Guiding to Prioritize Recursive Path. In order to limit the states explored and the memory used, GoSonar uses a DFS approach which can be guided by the path found in CFG. The CFG contains the path from the entry point of stem to the recursion itself and subsequently the path to execute the recursion as well i.e. the address of the basic blocks that needs to be visited in order to reach a certain target. This information can be used to create the priority among the successors of a state. Furthermore, GoSonar also limits the number of state being executed by limiting to one successor approach where only one state is explored at a time. This stops GoSonar from getting into state and memory explosion as well. If GoSonar fails to find a path

```
1  func (z *Reader) readHeader()(hdr Header, err error)
2  {
3    ...
4    if z.decompressor == nil {
5      z.decompressor = flate.NewReader(z.r)
6    }
7    ...
8  }
9
10 func (z *Reader) Read(p []byte) (n int, err error) {
11   ...
12   n, z.err = z.decompressor.Read(p)
13   ...
14 }
```

Figure 5: [compress/gzip/gunzip.go]
Public function containing unresolvable calls in `Read` for CVE-2022-30631

```
1 func Write(w io.Writer, order ByteOrder,
            data interface{}) error {
2   ...
3   case *int16:
4     order.PutUint16(bs, uint16(*v))
5   ...
6   v := reflect.Indirect(reflect.ValueOf(data))
7   ...
8   _, err := w.Write(buf)
9   return err
10 }
```

Figure 6: [encoding/binary.go]
Example of different types of function calls in Go

using the current state, another state is taken out of the stash until the stash is empty or the target is reached.

Indirect Call Resolver. A correct stem can help initialize the data structure, but often a sequence of disjoint function calls is needed. For example, in Figure 4, `Unmarshal` creates a `byteReader` and uses it to create a `Decoder` via `NewDecoder`. However, such a function may not always be available. In line 12 of Figure 5, a call is made to `Read` of the member `decompressor`. Here `decompressor` is an interface of type `io.ReadCloser`, which implements `Read` and `Close`. A concrete object for this interface is only assigned at line 5 under the function `readHeader`. Thus, to resolve the call in line 12 in the public function `Read`, a different private function must be called first.

GoSonar addresses this by using the source code to define a function's signature and search for matching functions in the binary. It first uses the function call's address to fetch the source code line number, possible with debug symbols. The source line is then parsed into AST to analyze the function call. GoSonar examines various Go AST expressions such as `SelectorExpr` (dot operation) and `StarExpr` (pointer operation) to determine the static type of a function call. The analysis aims to identify four pieces of information for a call instruction: the package name, the interface or data structure, and the function name. GoSonar must handle several cases to extract this information:

- **Interface Method Call** A function call can be made to an interface method that is resolved during runtime. This interface can be a member variable of a data structure that is assigned during the initialization of an object as seen in line 4 of Figure 5 where the variable `decompressor` can be any data structure that implements all the methods for `io.ReadCloser`. Moreover, it can be passed as a parameter to the function as shown on line 1 of Figure 6 where the variable `w` is of type interface `io.Writer`. Thus in this case, if GoSonar wants to resolve the function call on line 8 of Figure 6 it has to find the type of `w` for package name and interface name, whereas the function name can be parsed from the line directly.
- **Object Method Call** A data structure may contain multiple references to other data structures (as member variables) and make calls to public methods on them. For example, on line 4 of Figure 6 the object `order` of type `ByteOrder` is used to call the function `PutUint16`. GoSonar analyzes the type of the variable `order` statically to find the package and data structure, and the function name can be transcribed from the line itself.
- **Package Method Call** A package can have public functions that can be thought of as static functions in the OOP paradigm, which mostly work on the passed argument. An example of such a function call can be seen on line 6 of Figure 6 where the functions `ValueOf` and `Indirect` of the package `reflect` have been called. For such a call, GoSonar only parses the package name and the function name.
- **Chained Call** Often multiple calls can be made on a single line of code, leading to multiple call instructions on the same line of code. For example, on line 6 we can see that both functions `Indirect` and `ValueOf` have been called and, although they will have different instruction addresses, they point to the same line of code. GoSonar handles this by employing a LIFO queue for function calls. We observe that the innermost function of a function chain is called first on a binary level. Thus, upon first resolve request on a chain, resolves all the function calls and creates a queue. Upon subsequent calls, the rest of the functions are popped until the queue is empty. In our example, this will return `reflect.ValueOf` first and then on subsequent calls will return `reflect.Indirect`.

The overall design of the call resolver is shown in Figure 7 where addr2line is used on the address provided by the Go executable which provides the line number of the source code. The source code is then parsed with AST parser to extract the AST related to the call instruction. Finally, this AST is analyzed to extract the four components.

Matching Function Signature. Once the function name is resolved in terms of package, interface, data structure,

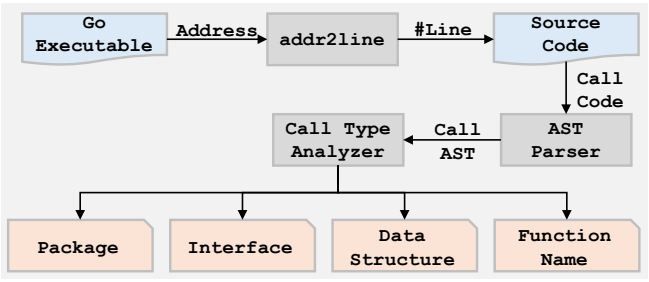

Figure 7: Overall Design for Indirect Call Resolver for Go using Source Code AST Parsing

and function name, GoSonar must identify the matching functions at the binary level. In the compiled version, all these details form the function name, such as `encoding..z2fbinary.Size`, where `encoding` is the package, `binary` is the sub-package, and `Size` is the function name. Ideally, each signature matches a single function, which is true for package public functions and data structure member methods, but not for interfaces. Using symbolic execution, it can resolve a call target to multiple addresses with constraints. For example, the `io.Writer` interface in line 1 of Figure 6 has a single function `Write`. Any structure implementing `Write` implements the interface, such as `bufio.Writer`, `bytes.Writer`, and `archive/tar.Writer`. Any of these can replace variable `w` in line 1. In such cases, GoSonar matches only the function name at the binary level, finding all the examples and leading symbolic execution to explore all paths.

This approach explores all data structures that implement this interface and finds all execution trees. If recursion is uncontrolled with a specific implementation, GoSonar can help infer the data structure needed to trigger the vulnerability. This is shown in Figure 3 – ❸, where symbolic execution is guided by the call resolver and path guidance, producing success and failure cases for the next step.

3.4. Constraint Reasoning

If symbolic execution reaches a specific program point, all collected constraints can be solved to find at least one solution. States with unsolvable or conflicting constraints are discarded. Should recursion be possible n times symbolically, at least one concrete input exists to achieve this. However, the primary challenge lies in determining whether the $n{+}1^{th}$ recursion is feasible after n successful recursions. A terminating condition must function as a constraint on the program state that evolves with each recursion, guiding the execution towards termination. The distinction between the constraints at $n-1^{th}$ and n^{th} recursion demonstrates the execution trajectory and the additional constraints imposed. The constraint analyzer reviews this difference to detect varying constraints that could serve as a termination condition. The lack of such a constraint indicates the uncontrolled nature of the targeted recursion. For failed scenarios, GoSonar organizes them by the execution stage and the reason for failure.

Constraint Analyzer. The difference between the constraints of consecutive calls is the set added since the last recursion. For example, take our motivating example in Figure 2 where the depth serves as a terminating condition when the patch works. The following equation shows the *intended* set of constraints on depth after visiting the function `decIgnoreOpFor` for three times.

$$\begin{aligned} depth == 0 \quad \&\& \\ depth < maxIgnoreNestingDepth \quad \&\& \\ depth + 1 < maxIgnoreNestingDepth \quad \&\& \\ (depth + 1) + 1 < maxIgnoreNestingDepth \end{aligned}$$

The first time the function is visited, the initial value of depth is 0 adding the first constraint. However, upon each revisit, an additional constraint is added on top of the previous constraints. The new constraint added for each iteration can be modeled as $depth + n * 1 < maxIgnoreNestingDepth$ where there is a fixed value of n until this constraint can be satisfied. This can be visualized with the help of a function $f(x_i)$ where x is a variable of the recursive function and $f(x_i)$ is the set of constraints collected on x after the i^{th} visit to the function. Now there are four cases that can happen, which are explained as follows:

(a) **Linear.** There can be a steady increase in the set of constraints as seen for `depth` in the fixed recursion. This is true for all monotonic stepping functions which constitute most of the recursive functions in modern programs. In such a case, if the delta of constraint is satisfiable indefinitely, then the recursion is uncontrolled, otherwise controlled.

(b) **Constant.** No new constraints need to be satisfied to continue the recursion. In our motivating example, for the missed recursion, the only fixed constraint is `depth == 0 && depth < maxIgnoreNestingDepth`. That is, when the larger recursion is executed, `depth` is reset to 0, resetting all constraints and making delta empty. These recursions are uncontrolled.

(c) **Logarithmic.** Constraints can change logarithmically, a rare case in real-world programs, and cannot be formally verified to terminate. For example, if `depth` increases by $1/i$ with each i^{th} recursion, the increment may never reach `maxIgnoreNestingDepth`. Such a recursion is controlled if the termination condition is below the convergence point; otherwise it is uncontrolled.

(d) **Oscillatory.** The last type is an oscillating function, where constraints fluctuate with recursion and cannot be formally verified to terminate. For example, if one function in a recursion increases `depth`, while another resets it.

```
1 func sizeof(t reflect.Type) int {
2   switch t.Kind() {
3     case reflect.Array:
4       if s := sizeof(t.Elem()); s >= 0 {
5         return s * t.Len()
6   }
7   ...
8 }
```

Figure 8: [encoding/binary.go]
Example of an unknown uncontrolled recursion detected with GoSonar

GoSonar focuses on the first two kinds of function that are more common in practise, as evident from Table 2 which shows the taxonomy of recursive functions in OSS and their constraint reasoning type. GoSonar gathers the delta $f(x_2) - f(x_1)$ and $f(x_3) - f(x_2)$ from 3 recursive executions, canonocalizes the memory addresses, and matches the constraints in these deltas. If the deltas can be matched, then it falls under category a i.e. the slope is constant. The delta is then analyzed for indefinite satisfiability to detect uncontrolled recursion. For category b, if there are 3 recursions with same set of constraint then the delta will be empty, which can be satisfied indefinitely by definition.

An example of *category a* for new vulnerability detected by GoSonar can be seen in Figure 8 where the only requirement for recursion to be uncontrolled is that `t.Kind()` must be `Array` for every nested element of `t`. Under the hood, the function `t.Kind` accesses a private integer variable `kind` that is matched to the constant integer value of `reflect.Array.kind`. On a binary level, the constraint becomes $t.kind == reflect.Array.kind$ where the only change on each recursion is the address of the object `t`. We can prove this constraint to be uncontrolled by canonicalizing the memory addresses, which makes the constraint indefinitely satisfiable.

Ranking Function. There are several steps of analysis by GoSonar that a lasso must go through before it can be identified as an uncontrolled recursion. GoSonar ranks the lassos that do not make it to the end depending on how far it executes. For example, the constraint reasoning may fail for a lasso if it is not in category a or b. On the other hand, there can be lassos that finish executing the stem but never finish a single recursion that have less chance of being uncontrolled. That is, GoSonar may miss some cases, which is true for any analysis for the detection of uncontrolled recursion. Thus, GoSonar categorizes failed cases into the following categories:

- **No Paths.** The underconstrained symbolic execution exhausted all possible paths before meeting the termination condition which is hitting the recursive functions for at least 3 times for 2 deltas required for constraint reasoning.
- **Timeout.** These lassos could not be analyzed within the timeout limit of 600 seconds.
- **Recursion Verified.** For these lassos the recursions were executed but the constraint could not be analyzed to be uncontrolled.
- **Symbex Errored.** For these lassos the symbolic execution threw an error and could not be completed.

GoSonar uses a central database and maintains a status for each lasso such that they can be later anlayzed by developers.

4. Evaluation

We evaluated GoSonar on standard library packages of Go that are used by all Go programs in some way. The binaries are compiled by statically linking libgo to facilitate analysis of the package. Furthermore, GoSonar focuses on a particular package by isolating the functions under that particular package. For verification of the found recursions, we have manually scrutinized them and created Proof of Concepts (PoCs) for them. These PoCs were then used to verify the uncontrolled recursion and subsequently the resource exhausting properties. In evaluating GoSonar, we try to answer the following research questions:

RQ1 How many recursions can GoSonar detect and what are the effects of each component of GoSonar on it?
RQ2 What are the new vulnerabilities GoSonar found and how many of them are exploitable?
RQ3 How accurate is GoSonar and how does it perform on previously known CVEs?
RQ4 How does GoSonar perform compared to state-of-the-art on a standard dataset?

4.1. Experiment Environment

We have implemented GoSonar by extending angr version 9.2.91 with our custom call resolver and exploration technique with Python version 3.12. GoSonar uses graph-tool version 2.59 for graph-based algorithms such as circuit detection and shortest path detection. We have analyzed 36 standard library packages from Go version 1.14.6. For building the binaries, we have built gcc 11.4.0 with gccgo from scratch with debug symbols. We have used Docker version 24.0.5 and Ubuntu:22.04 as the base for our docker containers to build and execute the PoC to verify the result found by GoSonar and to observe the resource exhausting power of the PoCs. The experiments were carried out on a single 10-Core 12th Gen Intel Core i7-12700 machine with 128GB memory and running Linux 5.15.0-84-generic x86_64. We have limited our analysis to lassos with at most five functions in loop or stem in total ten functions at most and analyzed each lasso for 600 seconds before labeling them as timeout, whereas the contemporary tools were given 900 seconds before labeling them as timeout.

4.2. Dataset

We focus on the logical vulnerabilities prevalent in Go. Table 1 lists the top 10 CWEs for Go with count

CWE	Count ↓	Description
CWE-22	41	Path Traversal
CWE-20	36	Improper Input Validation
CWE-400	35	Uncontrolled Resource Consumption
CWE-770	32	Allocation of Resources Without Limits or Throttling
CWE-532	16	Insertion of Sensitive Information into Log File
CWE-79	13	Cross-site Scripting
CWE-347	13	Improper Verification of Cryptographic Signature
CWE-295	13	Improper Certificate Validation
CWE-674	12	Uncontrolled Recursion
CWE-863	12	Incorrect Authorization
Total	**223**	

TABLE 1: Top 10 CWEs for Go with count and description

Category ↓			Count(%)	CR
Incorrect Recursion Design (1)	**Incorrect Arguments (1.1)**		7 (10.8%)	b
	Incorrect Return (1.2)		25 (38.5%)	a
	Deep Recursion (1.3)		33 (50.7 %)	a
Unexpected Recursion (2)	**Incorrect Self-invoking (2.1)**	**Misusing Namespace (2.1.1)**	38 (29.9%)	a/b
		Miscalling Inherited Method (2.1.2)		
		Misusing Method Overloading (2.1.3)		
		Missing Undef Instruction (2.1.4)		
	Incorrect Cyclic Invoking (2.2)		24 (18.9%)	a/b

TABLE 2: Taxonomy of uncontrolled recursion Found in OSS Projects [21] [CR: Constraint Reasoning Category (§3.4)]

and description. All 223 vulnerabilities in these classes are logical, some being umbrella terms, others more specific. CWE-674 (Uncontrolled Recursion) is a critical issue for Go and it directly contributes to other classes, including CWE-400 (Uncontrolled Resource Consumption) and CWE-770 (Allocation of Resources Without Limits or Throttling). Together, these issues make up 79/223 (35%) CVEs in Table 1. Under the broader category of resource consumption, CWE-674 is the only specific one with a well-defined cause, hence our focus. We believe that constraint reasoning can also detect other vulnerabilities.

To ensure GoSonar's soundness, we evaluated it on a dataset [21] of uncontrolled recursion in real-world programs. The dataset includes 445 nontermination bugs from 3,142 GitHub commits in popular OSS projects like Linux and Chromium. It categorizes 127 cases of uncontrolled recursion into 2 major and 8 specific categories, as shown in Table 2 and contains simplistic representative programs for each category, with patched and vulnerable versions.

The dataset also indicates the prevalence of each category and the type of constraint reasoning. All categories fall into either category a or b, which GoSonar can handle, demonstrating GoSonar's superior performance over contemporary tools. , were created to test contemporary tools and GoSonar. We evaluated the best tools for termination detection and nontermination proving. UAutomizer led the termination category at SV-COMP (2017-2021). In 2020 and 2021, CPAchecker and 2LS placed second and third. AProVE ranked in the top three (2017-2019). T2 outperformed Julia [22] and TNT [11].

4.3. Detailed Results

The evaluation of GoSonar is shown in Table 3. The table lists packages with at least one valid candidate, followed by lasso. Some packages have unreachable recursions, making them invalid.

Input. The table shows the binary sizes, which range from 6.76MB to 31.28MB due to static linking with libgo. The total size of all binaries is 179.27MB, with an average of 12.81MB per binary. The table next shows the number of nodes and edges for the CFG to Call Graph and lastly for the Pruned Call Graph. The CFG ranges from 59,535 nodes and 121,954 edges (reflect) to 246,325 nodes and 536,757 edges (net). The call graph drastically reduces this number to 11,378 nodes and 19,790 edges (reflect) to 48,416 nodes and 90,014 edges (net). GoSonar further prunes the call graphs to reduce the range to 703 nodes and 639 edges – 6,858 nodes and 11,972 edges.

Component Output. After the input, the table shows the output of GoSonar components. The first column shows the number of candidates found by the candidate finder (❶–§3.1), the second column shows how many can create lassos by verifying reachability with a stem (❷–§3.2), and the last column shows how many pass the filter (❸–§3.3). The table lists candidate recursions per package, ranging from 2 (index) to 823 (go). Not all candidates form valid lassos with a stem that can reach the recursion. Some recursions are reachable via multiple paths, producing multiple lassos per candidate recursion. The number of lassos ranges from 1 (compress) to 17,710 (go). In total, GoSonar identifies 983 loops and 20,036 lassos across all packages. After filtering, most packages have all lassos satisfying the requirement, except for go, where only 293 out of 17,710 lassos pass the filter. Overall, GoSonar analyzes 2,439 lassos from all packages.

Ablation Analysis. We perform an ablation study to assess the effect of each component of GoSonar. For each form of analysis, the table shows how many recursions can be symbolically executed at least once, proving the reachability of the recursion and how many can be verified to be uncontrolled. We detail the three forms of analysis that we have executed:

Constraint Reasoning (❶ + ❹). The first analysis approach uses only constraint reasoning with guided symbolic execution on candidates without verifying reachability, stem

		Input						Component			Analysis					
		CFG		CallGraph		Pruned		❶	❷	❸	Constraint Reasoning		CR w/ Stem Detection		GoSonar	
Package ↓	**Size(MB)**	**#Nodes**	**#Edges**	**#Nodes**	**#Edges**	**#Nodes**	**#Edges**	**#Cand.**	**#Lassos**	**#Filt.**	**#Exec.**	**#Ver.**	**#Exec.**	**#Ver.**	**#Exec.**	**#Ver.**
compress	11.26	92,235	188,989	18,830	32,879	1,966	2,692	2	1	1	0	0	0	0	1	1
database	10.84	88,413	182,738	18,189	32,067	1,801	2,662	2	8	8	0	0	0	0	0	0
encoding	14.18	118,806	252,480	23,346	42,101	2,791	4,832	37	173	173	0	0	0	0	2	1
fmt	10.02	82,370	168,695	16,876	29,411	1,579	2,267	46	1,760	1,580	0	0	0	0	207	0
go	21.04	174,880	377,940	34,131	62,211	4,401	7,493	823	17,710	293	1	1	10	9	13	9
html	13.18	108,241	226,141	21,816	38,982	2,492	4,019	5	6	6	0	0	0	0	0	0
index	8.55	75,410	155,142	14,254	24,790	1,269	1,306	2	2	2	0	0	0	0	0	0
math	11.68	96,570	199,606	19,597	34,511	2,156	3,213	7	145	145	0	0	6	2	6	2
net	31.28	246,325	536,757	48,416	90,014	6,858	11,972	11	23	23	0	0	0	0	0	0
path	10.03	82,401	167,065	16,853	29,182	1,585	2,022	2	2	2	1	1	1	1	1	1
reflect	6.76	59,535	121,954	11,378	19,790	703	639	7	30	30	0	0	0	0	1	0
sort	6.87	60,586	123,891	11,578	20,135	750	679	4	10	10	0	0	0	0	0	0
testing	11.59	95,708	197,256	19,809	34,885	2,143	3,019	3	7	7	0	0	0	0	0	0
text	11.99	98,199	202,718	20,127	35,509	2,196	3,313	32	159	159	1	1	1	1	52	1
Total	179.27	1,479,679	3,101,372	295,200	526,467	32,690	50,128	983	20,036	2,439	3	3	18	13	283	15
Average	12.81	105,691.36	221,526.57	21,085.71	37,604.79	2,335.00	3,580.57	70.21	1,431.14	174.21	-	-	-	-	-	-

TABLE 3: Detailed evaluation of GoSonar [Exec. = Lassos Executed Ver. = Lassos Verified Uncontrolled]
❶ Candidate Finder (§3.1) ❷ Reachability Verifier (§3.2) ❸ Symbolic Execution Guide (§3.3) ❹ Constraint Reasoning (§3.4)

detection, or call resolver. The table shows that very few recursions can be analyzed and verified in this way. These are the recursions in public functions which makes them reachable without any stem. Three such cases can be seen in packages go, path, and text all of which finish 3 recursion and can be proven to be uncontrolled with constraint reasoning, demonstrating the general applicability of constraint reasoning.

Constraint Reasoning with Stem Detection (❶ + ❷ + ❹). The next form of analysis uses stem detection on top of the last form of analysis, but does not utilize the call resolver. That is, it finds more recursions in private functions that can be reached via the public function. Furthermore, it also utilizes our insight into how public functions are chained together in Go programs undertaking necessary initialization to facilitate symbolic execution and improve the analysis result. This form of analysis can execute 18 lassos, of which 13 can be proven to be uncontrolled.

GoSonar. Lastly, the table shows how many lassos can be executed and verified using all the components of GoSonar, including the call resolver which relies on source code. The call resolver helps facilitate the symbolic execution by providing concrete values for indirect calls, which results in more lassos being executed for analysis. As the table shows, adding the call resolver greatly increases the number of lassos that can be executed. The number of lassos that can be executed for multiple recursions increases to 283, these serve as potential threats and can be scrutinized manually depending on how many recursion they complete.

Only the last form of analysis uses the call resolver, which relies on the source code that shows the applicability of GoSonar as a generalized binary analysis tool for the detection of uncontrolled recursions. Our insights and their effect on the analysis is reflected in the increase in the number of lassos analyzed in each step. The majority of lassos that could not be executed stem from underconstrained symbolic execution of a library and is an orthogonal problem to be solved. These lassos fall into the category *no path* or *symbex errored* category as mentioned in §3.4. However, as can be seen, GoSonar still finds and proves 15 uncontrolled recursions in all standard library packages of Go. Some of these are already found and have associated CVEs and some lassos originate from a single recursion where GoSonar finds multiple paths to trigger it. This analysis answers our **RQ1**, GoSonar finds 15 uncontrolled recursions where the addition of each component increases the overall effectiveness of GoSonar.

Uncontrolled Recursions Detected. Table 4 answers **RQ2** by showing all uncontrolled recursions detected by GoSonar, their code location, and novelty. The table lists the package and function name of each recursion. Some recursions produce multiple lassos, indicating multiple uncontrolled paths. The table also shows the recursion size (number of functions) and stem (1 to 5). It compares the time taken to detect these lassos with and without the symbolic execution guidance component (❸–§3.3), showing a reduction in time. GoSonar takes 838.92 seconds to execute all lassos for at least 3 recursions without guidance and 634.83 seconds with guidance. The average execution time decreases from 64.53 seconds to 42.32 seconds. Lastly, Table 4 shows whether the uncontrolled recursions are true and if they are new bugs. Most of the recursions identified by GoSonar are true positives verified by creating PoCs, except one. The false positive is a unique case explained in the limitation section. Overall GoSonar finds and proves

Package	Func. Name	#Funcs R	#Funcs S	Exec. Time (s) w/o ❸	Exec. Time (s) w/ ❸	Ver.?	New?
compress	glob/Reader.Read	1	1	-	1.01	✓	✗
encoding	binary.sizeof	1	3	-	12.18	✓	✓
go	filterExprList	2	5	465.35	357.36	✓	✓
			4	117.51	46.68	✓	✓
			4	17.98	26.38	✓	✓
			3	26.79	8.72	✓	✓
			4	36.79	18.83	✓	✓
	Sign	1	1	69.10	62.38	✗	-
	appendReverse	1	3	35.04	24.23	✓	✓
			3	1.62	1.49	✓	✓
			3	1.32	1.40	✓	✓
math	mulRange	1	2	2.18	1.06	✓	✓
			3	30.62	29.75	✓	✓
path	Glob	1	1	33.18	41.97	✓	✗
text	IsEmptyTree	1	1	1.45	1.40	✓	✓
Total		8	9	41	838.92	634.83	14	13
Average					64.53	42.32		

TABLE 4: Uncontrolled recursions identified by GoSonar in standard library packages of Go

13 new lassos to be uncontrolled which represents the 5 vulnerabilities it finds.

Detection of Known Vulnerabilities. Table 5 answers **RQ3**, that is how GoSonar handles the previously detected vulnerabilities which pose a real security threat. As seen, among the 6 cases GoSonar can identify all of them, however, GoSonar fails to verify that they are uncontrolled recursion in nature. The ones that can not be verified is due to limitations of symbolic execution and call resolver of GoSonar. An example of a call that cannot be resolved with GoSonar is explained in limitation section. Angr uses an interface for the backend constraint solver claripy which can use among others Z3 constraint solver. However, sometimes the constraints become complex in nature and angr throws an error in such case GoSonar can not continue the analysis.

4.4. Comparative Analysis

In order to answer **RQ4** we have evaluated GoSonar against several state-of-the-art tools on a standard dataset populated from patched nontermination bugs in OSS projects caused by infinite recursion. Table 6 shows the result of the contemporary tools along with GoSonar. The dataset does not have any representative program for categories 2.1.1 and 1.3 due to the complexity of these categories. However, GoSonar shows that it can detect 1.3 or deep recursions in the primary evaluation. For each category, the dataset contains a representative vulnerable version (V) and patched version (P) of a program, as shown in Table 6 with Type. The table has four outcomes for each case, *correct detection*, *maybe*, *unknown*, and *errored*. Given the nature of the problem, contemporary tools can produce inconclusive results in their own way, which have been represented in the dataset as maybe (?). When tools cannot analyze a certain program, it is represented with unknown (UN), and finally if it crashes or hits timeout, it is represented with errored (E).

CVE	Package	Function	Identified?	Verified?
CVE-2022-1962	go	Parser	✓	✗
CVE-2022-28131	encoding/xml	Decoder.Skip	✓	✗
CVE-2022-30632	path/filepath	Glob	✓	✓
CVE-2022-30633	encoding/xml	Unmarshal	✓	✗
CVE-2022-30631	compress/gzip	Reader.Read	✓	✓
CVE-2022-30635	encoding/gob	Decoder.Decode	✓	✗

TABLE 5: Previously detected uncontrolled recursions in standard library packages of Go version 1.14.6 or earlier

Correct detection means identifying a recursion as uncontrolled in the vulnerable version and controlled in the patched version. As can be seen in Table 6, GoSonar outperforms all contemporary tools in all categories by correctly detecting both the presence (vulnerable) and the absence (patched) of uncontrolled recursion. As seen in the table, there are very few cases in which contemporary tools successfully analyze and detect vulnerability. Furthermore, for categories 2.1.2 and 2.1.3, these can only be possible for C++ programs that none of the tools handle. GoSonar on the other hand, uses binary, removes such dependency, and analyzes them without any issues. The dataset identifies several points for the failure of contemporary tools, among which the leading reasons for producing *unknown* are pointer manipulation, arrays, and data structures. However, modern programming relies heavily on these features which makes these tools futile for real world programs. The only modification in GoSonar to analyze these binaries was that (i) the call resolver component could not be used due to its applicability, and (ii) stem detection used `main` as the only acceptable starting point for lassos.

5. Discussion

This section discusses the limitations of GoSonar along with how they can be addressed in future work of their own. These limitations pose a limited obstacle while requiring significant investigation in order to be addressed and thus are left out of scope of this work.

5.1. Limitation

Symbolic execution, especially an underconstrained one on a heavily instrumented program, creates difficulty in analysis. There are two major limitations we have faced that could not be solved with GoSonar, which are:

Unresolveable Function Call. As mentioned earlier, a key contribution of GoSonar is resolving indirect function calls by incorporating the source code. However, there is

#	Type	UAutomizer	AProVE	CPAchecker	2LS	T2	GoSonar
1.1	V	✓	?	UN	E	E	✓
	P	✓	?	✓	E	E	✓
1.2	V	E	?	UN	UN	E	✓
	P	E	?	UN	✓	E	✓
2.1.2	V	-	-	-	-	-	✓
	P	-	-	-	-	-	✓
2.1.3	V-1	-	-	-	-	-	✓
	P-1	-	-	-	-	-	✓
	V-2	-	-	-	-	-	✓
	P-2	-	-	-	-	-	✓
	V-3	-	-	-	-	-	✓
	P-3	-	-	-	-	-	✓
	V-4	-	-	-	-	-	✓
	P-4	-	-	-	-	-	✓
2.1.4	V	✓	✓	UN	UN	E	✓
	P	UN	✓	✓	✓	E	✓
2.2	V-1	✓	?	UN	E	E	✓
	P-1	✓	✓	✓	✓	E	✓
	V-2	UN	?	UN	UN	E	✓
	P-2	UN	?	UN	UN	E	✓

TABLE 6: Comparative analysis of GoSonar
✓= Correct UN = Unknown E = Errorred ? = Maybe
V = Vulnerable/Non Terminating P = Patched/Terminating

```
1 type byteReader struct {
2   fmt.ScanState
3 }
4
5 func (r byteReader) ReadByte() (byte, error) {...}
6
7 func (r byteReader) UnreadByte() error {...}
```

Figure 9: [math/big/intconv.go]
Unresolveable function call in Go containing data structure wrapper

a case where GoSonar fails to resolve a call due to the design of Go. Figure 9 shows such a case where the data structure `byteReader` on line 1 is used as a wrapper for data structure `fmt.ScanState` to extend it by implementing `ReadByte` and `UnreadByte` functions. The current implementation of call resolver points to line 2, however no function call is present to be analyzed by GoSonar.

Control by Type. Second, we can see a failed case in Figure 10, the only false positive for GoSonar where there is a recursive call to function `Sign` if the input is a complex number. However, in order for it to be uncontrolled, either the real part (`re`) or the imaginary part (`im`) has to be a nested complex number as well, which by definition is not possible. Furthermore, the code also ensures that anyone outside of this package can not implement the parent interface of `complexVal` – `Value` by incorporating a private function in the interface. All of this restricts the way to create a complex number or a `complexVal` and thus GoSonar finds this to

```
1 // A Value represents the value of a Go constant.
2 type complexVal struct{ re, im Value }
3
4 func Sign(x Value) int {
5   switch x := x.(type) {
6     ...
7     case complexVal:
8       return Sign(x.re) | Sign(x.im)
9     ...
10 }
```

Figure 10: [go/constant/value.go]
Example of a recursion that can not be uncontrolled by design enforced by type

be an uncontrolled recursion, whereas there is a hidden termination condition in a broader context.

5.2. Future Work

GoSonar establishes the foundation for several future works in the detection of logical vulnerabilities in memory safe language. These works can address the limitation of GoSonar or build on the idea of constraint reasoning for different kinds of logical vulnerability.

Data Structure Initialization. A key contribution of GoSonar is demonstrating that a proper stem can lead to data structure initialization necessary for triggering vulnerabilities. However, finding a function that calls all necessary functions with concrete values is challenging. Future work for GoSonar includes finding a chain of functions that initialize the data structure required to trigger vulnerabilities, possibly using code completion ML models to find preamble code for calling a target function.

PoC Generation. One insight we have while working for GoSonar is the lack of PoC for Go vulnerabilities. Although every vulnerability fixed comes with a testing function to ensure that the patch addresses the vulnerability, a PoC serves a proof of the existence of a vulnerability before a patch is released. Furthermore, PoCs also help developers guide to the point of manifestation and facilitates detection of root cause. Thus, automatic PoC generation can be possible based on the constraints gathered to reach a point in the program execution.

Control by Type Detection. One of the limitations of GoSonar is the lack of a way to detect control by type where a recursion's termination condition is enforced by design. Modeling such a case would require similar efforts to type confusion detection such as building a class tree where the constraints can not be satisfied under the current class tree. For example, in our example of the function `Sign` if we can detect all the public functions that can create a `complexVal` we can make a class tree which would enforce that the real or imaginary part of a complex number can not be of type `complexVal` itself.

6. Related Work

Nontermination is a heavily studied problem; however, very few works provide a practical tool that can be used on real-world programs. In program verification realm, the datasets used are often synthetic, which encompasses various kinds of recursive function. Among the very few works aimed and evaluated against real-world programs, we find that they are either evaluated against a representative version of the vulnerability or use fuzzer to generate test cases needed to verify the initial finding. Overall, the current body of work on nontermination detection can be mainly divided into two categories:

Termination Detection. The majority of termination detection employs some kind of ranking function like Proteus [25] which summarizes the loop in order to detect termination. On the other hand, Ultimate Automizer or UAutomizer [12] abstracts all the program trace to automata and then unites them for analysis with a ranking function. Lastly, one work [28] uses ML model to learn the bounds of a loop whose by product can be learning the termination condition and subsequently termination detection.

Non Termination Detection. One of the oldest work TNT [11] proves the non termination by generating counterexample of termination based on revisit of recurrent set of states. Whereas other works [7] focused on simplistic integer programs with non-determinism and determines nontermination by reversal of a program's transition system. Continuation of Proteus, Loopster [26] tried to reduce nontermination to a reachability problem and finally used path dependency automaton [27] to capture dependency among several diverging paths of a loop. More practical solutions such as 2LS [19] use inter-procedural abstract interpretation for a modular termination analysis for C programs. AProVE [8] is the only work that uses constraint solving; however, they used it to find a recurrent set in a loop to determine nontermination. Lastly, only one work [29] works on real-world programs and successfully finds bugs in C/C++ programs; however, it specifically focuses on infinite loops where the termination condition is decided by a primitive data type. Moreover, it uses fuzzing for 240 hours to generate hang test cases, which are then analyzed to detect such nontermination cases.

All of these works provide a basis for detection of nontermination based on different forms of analysis of the program where a lot of assumptions and restrictions are put on the program. This prevents these works from being applied directly to real-world programs to analyze within a reasonable amount of time. GoSonar is a work first of its kind that uses a real-world dataset along with an evaluation of the standard library of a complicated language like Go.

7. Conclusion

In conclusion, the proliferation of safe memory languages has redirected attention towards logical vulnerabilities and their corresponding attack surfaces. The inherent complexity in modeling and detecting these logical vulnerabilities has forced contemporary tools to hypothesize solutions that are often impractical for real-world applications. GoSonar introduces an innovative methodology for modeling a specific logical vulnerability, namely uncontrolled recursion, through the analysis of a first-order derivative of the constraints at a particular execution point. GoSonar has demonstrated its efficacy by surpassing all contemporary tools on a standard dataset, while also incorporating techniques to scrutinize Go programs. GoSonar has identified 5 novel vulnerabilities within the standard library of Go, which significantly impact every Go program developed. We anticipate that GoSonar will become an integral component of the Go package vetting pipeline and, in the future, will facilitate the modeling of additional logical vulnerabilities that present challenges for detection by contemporary methodologies.

Acknowledgments

We would like to express our sincere gratitude to the anonymous reviewers for their constructive and insightful comments, which helped us improve the quality and clarity of this paper. We also thank the shepherd for their valuable guidance and support throughout the revision process. This research was supported in part by the ARO award W911NF2110081 and the National Security Agency under Grant H98230-22-1-0333. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] AddressSanitizer: A fast address sanity checker | USENIX. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[2] Fact sheet: ONCD report calls for adoption of memory safe programming languages and addressing the hard research problem of software measurability | ONCD. [Online]. Available: https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/memory-safety-fact-sheet/

[3] The urgent need for memory safety in software products | CISA. [Online]. Available: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

[4] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," vol. 41, no. 6, pp. 158–168. [Online]. Available: https://dl.acm.org/doi/10.1145/1133255.1134000

[5] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 184–190.

[6] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman, "T2: temporal property verification," in *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*. Springer, 2016, pp. 387–393.

[7] K. Chatterjee, E. K. Goharshady, P. Novotný, and D. Žikelić, "Proving non-termination by program reversal," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. Association for Computing Machinery, pp. 1033–1048. [Online]. Available: https://dl.acm.org/doi/10.1145/3453483.3454093

[8] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski *et al.*, "Proving termination of programs automatically with aprove," in *Automated Reasoning: 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings 7*. Springer, 2014, pp. 184–191.

[9] Golang Contributors, "encoding/gob: stack exhaustion in decoder.decode (cve-2024-34156)," https://github.com/golang/go/issues/69139, 2023, accessed: 2024-09-24.

[10] Google. (2024) Go use cases. [Online]. Available: https://go.dev/solutions/case-studies

[11] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '08. Association for Computing Machinery, pp. 147–158. [Online]. Available: https://dl.acm.org/doi/10.1145/1328438.1328459

[12] M. Heizmann, J. Hoenicke, and A. Podelski, "Termination analysis by learning terminating programs," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 797–813.

[13] D. Honnef. (2024) Staticcheck - the advanced go linter. [Online]. Available: https://staticcheck.dev/

[14] M. Michalik, "C vs rust vs go: Performance analysis," Jul 2019. [Online]. Available: https://medium.com/@marek.michalik/c-vs-rust-vs-go-performance-analysis-945ab749056c

[15] oscar6echo. (2024) Rust vs. c vs. go runtime speed comparison. [Online]. Available: https://github.com/oscar6echo/rust-c-go-speed

[16] owasp. (2023) Owasp top 10 api security risks – 2023. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0x11-t10/

[17] Praetorian. (2022) A static analysis tool for securing go code. [Online]. Available: https://github.com/praetorian-inc/gokart

[18] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, "HeapCheck: Low-cost hardware support for memory safety," vol. 19, no. 1, pp. 10:1–10:24. [Online]. Available: https://dl.acm.org/doi/10.1145/3495152

[19] P. Schrammel and D. Kroening, "2ls for program analysis: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*. Springer, 2016, pp. 905–907.

[20] securego. (2024) Go security checker. [Online]. Available: https://github.com/securego/gosec

[21] X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li, "Large-scale analysis of non-termination bugs in real-world OSS projects," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. Association for Computing Machinery, pp. 256–268. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3549129

[22] F. Spoto, F. Mesnard, and É. Payet, "A termination analyzer for java bytecode based on path-length," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 3, pp. 1–70, 2010.

[23] M. Strehovský. (2024) Compare binary sizes of canonical hello world in 17 different languages. [Online]. Available: https://github.com/MichalStrehovsky/sizegame

[24] K. V. (2018) How a go program compiles down to machine code. [Online]. Available: https://getstream.io/blog/how-a-go-program-compiles-down-to-machine-code/

[25] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: computing disjunctive loop summary via path dependency analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. Association for Computing Machinery, pp. 61–72. [Online]. Available: https://dl.acm.org/doi/10.1145/2950290.2950340

[26] X. Xie, B. Chen, L. Zou, S.-W. Lin, Y. Liu, and X. Li, "Loopster: static loop termination analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. Association for Computing Machinery, pp. 84–94. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3106260

[27] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le, and X. Li, "Automatic loop summarization via path dependency analysis," vol. 45, no. 6, pp. 537–557, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8241837?casa_token=G98C3foPXe4AAAAA:JOHFEbeFPP7gnLMD2Siap_gJmA1hfXxVnVnvv2CnmjG_GJYw-ZxGluWjEX6GthsIqRTjlmRv

[28] R. Xu, J. Chen, and F. He, "Data-driven loop bound learning for termination analysis," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. Association for Computing Machinery, pp. 499–510. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510220

[29] Y. Zhang, X. Xie, Y. Li, S. Chen, C. Zhang, and X. Li, "Endwatch: A practical method for detecting non-termination in real-world software," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 686–697.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper presents GoSonar, a static analysis tool for identifying uncontrolled recursions in Go programs. GoSonar identifies potential recursions through callgraph analysis, and then uses inductive constraint reasoning to determine whether they are in fact infinite recursions. GoSonar has detected 5 new vulnerabilities in the Go standard library.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Addresses a Long-Known Issue

A.3. Reasons for Acceptance

1) This paper presents a new static analysis tool to detect uncontrolled recursions in Go programs. This is an important problem, and GoSonar is able to outperform contemporary tools in finding these bugs in complex settings.
2) The paper provides a valuable step forward in an established field by enhancing the capabilities of currently available tools in identifying bugs in Go programs. GoSonar has identified 5 new vulnerabilities in the Go standard library.

A.4. Noteworthy Concerns

1) The evaluation mainly focuses on tools used for identifying uncontrolled recursions in Go programs. However, there are many previous works that focus on solving this problem for other programming languages, and it is unclear whether GoSonar is conceptually different from them or an adaptation of their ideas for Go programs.