Uncovering the Hidden Dangers: Finding Unsafe Go Code in the Wild

Johannes Lauinger[†], Lars Baumgärtner[*], Anna-Katharina Wickert[*], Mira Mezini[*]
Technische Universität Darmstadt, D-64289 Darmstadt, Germany
[*] *E-mail: {baumgaertner, wickert, mezini}@cs.tu-darmstadt.de*
[†] *E-mail: jlauinger@seemoo.tu-darmstadt.de*

Abstract—**The Go programming language aims to provide memory and thread safety through measures such as automated memory management with garbage collection and a strict type system. However, it also offers a way of circumventing this safety net through the use of the *unsafe* package. While there are legitimate use cases for *unsafe*, developers must exercise caution to avoid introducing vulnerabilities like buffer overflows or memory corruption in general. In this work, we present *go-geiger*, a novel tool for Go developers to quantify *unsafe* usages in a project's source code and all of its dependencies. Using *go-geiger*, we conducted a study on the usage of *unsafe* in the top 500 most popular open-source Go projects on GitHub, including a manual analysis of 1,400 code samples on how *unsafe* is used. From the projects using Go's module system, 38% directly contain at least one *unsafe* usage, and 91% contain at least one *unsafe* usage in the project itself or one of its transitive dependencies. Based on the usage patterns found, we present possible exploit vectors in different scenarios. Finally, we present *go-safer*, a novel static analysis tool to identify dangerous and common usage patterns that were previously undetected with existing tools.**

Index Terms—**Golang, Static Analysis, Memory Corruption.**

1. Introduction

Programming languages with direct memory access through pointers, such as C/C++, suffer from the dangers of memory corruption, including buffer overflows [1], [2] or *use-after-free* of pointers. Microsoft, e.g., reports that memory safety accounts for around 70% of all their bugs[1]. To avoid these dangers, many programming languages, such as Java, Rust, Nim, or Google's Go, use automatic memory management and prevent using low-level memory details like pointers in favor of managed object references. Thus, these languages are memory safe, eliminating most memory corruption bugs. However, there are valid use cases for such low-level features. Safe languages therefore provide, to varying degrees, escape hatches to perform potentially unsafe operations. Escape hatches may be used for optimization purposes, to directly access hardware, to use the foreign function interface (FFI), to access external libraries, or to circumvent limitations of the programming language.

However, escape hatches may have severe consequences, e.g., they may introduce vulnerabilities. This is especially problematic when *unsafe* code blocks are introduced through third-party libraries, and thus are not directly obvious to the application developer. Indeed, a recent study shows that unsafe code blocks in Rust are often introduced through third-party libraries [3]. Therefore, security analysts, developers, and administrators need efficient tools to quickly evaluate potential risks in their code base but also the risks introduced by code from others.

In this paper, we investigate Go and the usage of *unsafe* code blocks within its most popular software projects. We developed two specific tools for developers and security analysts. The first one, called *go-geiger* (Section 2.2) analyzes a project including its dependencies for locating usages of the *unsafe* API and scoring *unsafe* usages in Go projects and their dependencies. It is intended to give a general overview of *unsafe* usages in a project.

As *unsafe* usages are benign when used correctly, safe usages of *unsafe* exist. However, we identified several commonly used *unsafe* patterns, e.g., to cast slices and structs, which can break memory safety mechanisms. They introduce potential vulnerabilities, e.g., by allowing access to additional memory regions. We provide insights into the dangers and possible exploit vectors to these patterns, indicating the severe nature of these bugs leading to information leaks or code execution (Section 3.1).

While the Go tool chain provides a linter, called *go vet*, covering invalid *unsafe* pointer conversions, the linter fails to flag the potentially insecure usages. Thus, to support developers we implemented a second tool *go-safer* (Section 3.2) covering two types of those.

With the help of *go-geiger*, we performed a quantitative evaluation of the top 500 most-starred Go projects on GitHub to see how often *unsafe* is used in the wild (Section 4.2). Including their dependencies, we analyzed more than 62,000 individual packages. We found that 38% of projects contain *unsafe* usages in their direct application code, and 91% of projects contain *unsafe* usages either in first-party or imported third-party libraries.

We also created a novel data set with 1,400 labeled occurrences of *unsafe*, providing insights into the motivation

[1] https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

978-1-6654-0392-4/20/$31.00 ©2020 IEEE
DOI 10.1109/TrustCom50675.2020.00063

for introducing *unsafe* in the source code in the first place (Section 4.3). Finally, we used *go-safer* to find instances of our identified dangerous usage patterns within the data set. So far, in the course of this work we submitted 14 pull requests to analyzed projects and libraries, fixing over 60 individual potentially dangerous *unsafe* usages (Section 4.4).

In this paper, we make the following contributions:

- *go-geiger*, a first-of-its-kind tool for detecting and scoring *unsafe* usages in Go projects and their dependencies,
- a novel static code analysis tool, *go-safer*, to aid in identifying potentially problematic *unsafe* usage patterns that were previously uncaught with existing tools,
- a quantitative evaluation on the usage of *unsafe* in 343 top-starred Go projects on GitHub,
- a novel data set with 1,400 labeled occurrences of *unsafe*, providing insights into what is being used in real-world Go projects and for what purpose, and
- evidence on how to exploit *unsafe* usages in the wild.

2. Scanning for Usages of Go's *unsafe* Package

In this section, we give a brief introduction to *unsafe* in Go and then present our novel standalone tool *go-geiger* to identify *unsafe* usages in a project and its dependencies. Thus, it supports auditing a project and perhaps selecting dependencies more carefully.

2.1. Go's *unsafe* Package

The Go programming language, like other memory-safe languages, provides an *unsafe* package[2], which offers (a) the functions *Sizeof*, *Alignof*, and *Offsetof* that are evaluated at compile time and provide access into memory alignment details of Go data types that are otherwise inaccessible, and (b) a pointer type, *unsafe.Pointer*, that allows developers to circumvent restrictions of regular pointer types.

One can cast any pointer to/from *unsafe.Pointer*, thus enabling casts between completely arbitrary types, as illustrated in Listing 1. In this example, *in.Items* is assigned to a new type (*out.Items*) in Line 3 without reallocation for efficiency reasons. Furthermore, casts between *unsafe.Pointer* and *uintptr* are also enabled, mainly for pointer arithmetic. A *uintptr* is an integer type with a length sufficient to store memory addresses. However, it is not a pointer type, hence, not treated as a reference. Listing 2 presents an example of casts involving *uintptr*. In Line 2, the *unsafe.Pointer* is converted to *uintptr*. Thus, the memory address is stored within a non-reference type. Hence, the back-conversion in Line 3 causes the *unsafe.Pointer* to be hidden from the *escape analysis (EA)* which Go's garbage collector uses to determine whether a pointer is local to a function and can be stored in the corresponding stack frame, or whether it can *escape* the function and needs to be stored on the heap [4]. Storing the address of a pointer in a variable of *uintptr* type and then converting it back causes the *EA* to miss the chain of references to the underlying value in memory. Therefore, Go will assume a value does not escape when it actually does, and may place it on the stack. Correctly used it can improve efficiency because deallocation is faster on the stack than on the heap [4]. However, used incorrectly it can cause security problems as shown later in Section 3.1.

Listing 1: In-place cast using the *unsafe* package from the Kubernetes *k8s.io/apiserver* module with minor changes.

```
1 func autoConvert(in *PolicyList, out *audit.
      ↪ PolicyList) error {
2     // [...]
3     out.Items = *(*[]audit.Policy)(unsafe.
          ↪ Pointer(&in.Items))
4     return nil
5 }
```

Listing 2: Hiding a value from escape analysis from the *modern-go/reflect2* module.

```
1 func NoEscape(p unsafe.Pointer) unsafe.Pointer {
2     x := uintptr(p)
3     return unsafe.Pointer(x ^ 0)
4 }
```

2.2. *go-geiger*: Identification of Unsafe Usages

To identify and quantify usages of *unsafe* in a Go project and its dependencies, we developed *go-geiger*[3]. Its development was inspired by *cargo geiger*[4], a similar tool for detecting unsafe code blocks in Rust programs.

Figure 1 shows an overview of the architecture of *go-geiger*. We use the standard parsing infrastructure provided by Go to identify and parse packages including their dependencies based on user input. Then, we analyze the AST, which enables us to identify different usages of *unsafe* and their context as described in the next paragraph. Finally, we arrange the packages requested for analysis and their dependencies in a tree structure, sum up *unsafe* usages for each package individually, and calculate a cumulative score including dependencies. We perform a deduplication if the same package is transitively imported more than once. The *unsafe* dependency tree, usage counts, as well as identified code snippets, are presented to the user.

We detect all usages of methods and fields from the *unsafe* package, specifically: *Pointer*, *Sizeof*, *Offsetof*, and *Alignof*. Furthermore, because they often are used in unsafe operations, we also count occurrences of *SliceHeader* and *StringHeader* from the *reflect* package, and *uintptr*. All of these usages are referred to as *unsafe* usages in this paper. Additionally, we determine the context in which the *unsafe* usage is found, i.e., the type of statement that includes the *unsafe* usage. In *go-geiger* we distinguish between assignments (including definitions of composite literals and return statements), calls to functions, function parameter declarations, general variable definitions, or other not further specified usages. We determine the context by looking up in the AST starting from the node representing the *unsafe* usage, and identifying the type of the parent node.

[2] https://golang.org/pkg/unsafe

[3] https://github.com/jlauinger/go-geiger

[4] https://github.com/rust-secure-code/cargo-geiger

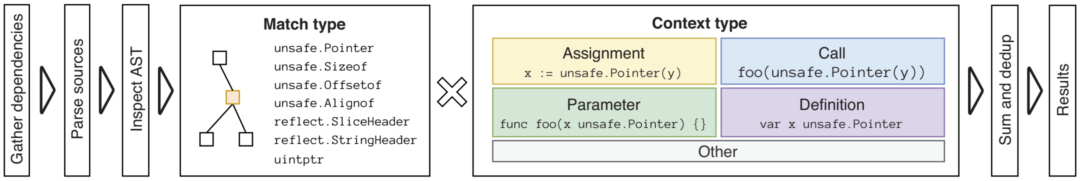

Figure 1: Architecture of *go-geiger* tool to detect *unsafe* usages

Listing 3: Conversion from string to bytes using *unsafe*

```
1 func StringToBytes(s string) []byte {
2     strHeader := (*reflect.StringHeader)(unsafe.
          ↪ Pointer(&s))
3     bytesHeader := reflect.SliceHeader{
4         Data: strHeader.Data,
5         Cap:  strHeader.Len,
6         Len:  strHeader.Len,
7     }
8     return *(*[]byte)(unsafe.Pointer(&
          ↪ bytesHeader))
9 }
```

3. Identifying Insecure Usages of *unsafe*

In this section, we present problematic code snippets including exploit information that we identified. Further, we introduce our linter *go-safer* to identify two known potentially dangerous *unsafe* patterns for slice and struct casts.

3.1. Potential Usage and Security Problems

In the following, we discuss potential threat models and exploit vectors against real-world *unsafe* Go code. We present a code pattern in Listing 3 that is very common in popular open-source Go projects (cf. Section 4). It is used to convert a string to a byte slice without copying the data. As in Go strings essentially are read-only byte slices, this is commonly done by projects to increase efficiency of serialization operations. Internally, each slice is represented by a data structure that contains its current length, allocated capacity, and memory address of the actual underlying data array. The *reflect* header structures provide access to this internal representation. In Listing 3 this conversion is done in Lines 2, 3, and 8 respectively. First, an *unsafe.Pointer* is used to convert a string to a *reflect.StringHeader* type. Then, a *reflect.SliceHeader* instance is created and its fields are filled by copying the respective values from the string header. Finally, the slice header object is converted into a slice of type *[]byte*.

Implicit Read-Only. The conversion pattern shown in Listing 3 is efficient as it directly casts between *string* and *[]byte* in-place. Using *bytes := ([]byte)(s)* for the conversion would make the compiler allocate new memory for the slice header as well as the underlying data array. However, the direct cast creates an implicitly read-only byte slice that can cause problems, as described in the following. The Go compiler will place strings into a constant data section of the resulting binary file. Therefore, when the binary is loaded into memory, the *Data* field of the string header may contain an address that is located on a read-only memory page. Hence, strings in Go are immutable by design and mutating a string causes a compiler error. However, when casting a string to a *[]byte* slice in-place, the resulting slice loses the explicit read-only property, and thus, the compiler will not complain about mutating this slice although the program will crash if done so.

Garbage Collector Race. Go uses a concurrent mark-and-sweep *garbage collector (GC)* to free unused memory [5]. It is triggered either by a certain increase of heap memory usage or after a fixed time. The GC treats pointer types, *unsafe.Pointer* values, and slice/string headers as references and will mark them as still in use. Importantly, string/slice headers that are created manually as well as *uintptr* values are not treated as references. The last point, although documented briefly in the *unsafe* package, is a major pitfall. Casting a *uintptr* variable back to a pointer type creates a potentially dangling pointer because the memory at that address might have already been freed if the GC was triggered right before the conversion.

Although not directly obvious, Listing 3 contains such a condition. Because the *reflect.SliceHeader* value is created as a composite literal instead of being derived from an actual slice value, its *Data* field is not treated as a reference if the GC runs between Lines 3 and 8. Thus, the underlying data array of the *[]byte* slice produced by the conversion might have already been collected. This creates a potential *use-after-free* or buffer reuse condition that, even worse, is triggered non-deterministically when the GC runs at just the ”right” time. Therefore, this race condition can crash the program, create an information leak, or even potentially lead to code execution. Figure 2 shows a visualization of the casting process that leads to the problems described here. The original slice is being cast to a string via some intermediate representations. The slice header is shown in green (at memory position 1), while the underlying data array (memory position 2) is shown in red. When the resulting string header (shown in blue at memory position 3) is created, it only has a weak reference to the data, and when the GC runs before converting it to the final string value, the data is already freed.

Escape Analysis Flaw. A third problem found in Listing 3 is that the *escape analysis (EA)* algorithm can not infer a connection between the string parameter *s* and the resulting byte slice. Although they use the same underlying data array, the EA misses this due to the fact that the intermediate representation as a *uintptr* is not treated as a reference type. This can cause undefined behavior if the

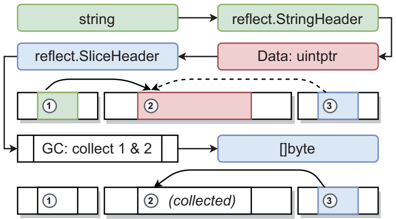

Figure 2: GC race and escape analysis flaw

Listing 4: Escape analysis flaw example

```
1  func main() {
2      bytesResult := GetBytes()
3      fmt.Printf("main: %s\n", bytesResult)
4  }
5
6  func GetBytes() []byte {
7      reader := bufio.NewReader(strings.NewReader(
         ↪ "abcdefgh"))
8      s, _ := reader.ReadString('\n')
9      out := StringToBytes(s)
10     fmt.Printf("GetBytes: %s\n", out)
11     return out
12 }
```

returned value from the casting function is used incorrectly. Listing 4 shows a program that uses the conversion function presented earlier (Listing 3). In the *main* function, *GetBytes* is called (Line 2), which creates a string and turns it into a byte slice using the conversion function. Within the *GetBytes* function, we create the string using a *bufio* reader similarly to if it were user-provided input. After the cast, *GetBytes* prints the resulting bytes (Line 10) and returns them to *main*, which also prints the bytes (Line 3). Although one might assume that both print statements result in the same string to be displayed, the second one in *main* fails and prints invalid data.

Because the string *s* is allocated in *GetBytes* the Go EA is triggered. It concludes that *s* is passed to *StringToBytes* and the EA transitively looks into that function. Here, it fails to connect *s* to the returned byte slice as described previously. Therefore, the EA concludes that *s* does not escape in *StringToBytes*. As it is not used after the call in *GetBytes*, the EA algorithm incorrectly assumes that it does not escape at all and places *s* on the stack. When *GetBytes* prints the resulting slice, the data is still valid and the correct data is printed, but once the function returns to *main*, its stack is destroyed. Thus, *bytesResult* (Line 2) is now a dangling pointer into the former stack of *GetBytes* and, therefore, printing data from an invalid memory region.

Code Execution. To show the severity of the issues identified above and that they are not just of theoretical nature, e.g., resulting in simple program crashes, we created a proof of concept for a code execution exploit using *Return Oriented Programming (ROP)* on a vulnerable *unsafe* usage. The sample incorrectly casts an array on the stack into a slice without constricting it to the proper length. This vulnerability causes a buffer overflow which we use to overwrite the stored return address on the stack, thus, changing the control flow of the program. Since Go programs are typically statically linked with a big runtime, there is a large number of ROP-gadgets available within the binary itself. We use gadgets to set register values and dispatch to system calls. Using the *mprotect* syscall, we set both the writable and executable permission bits on a memory page that is mapped to the program, and store an exploit payload provided via standard input there using the *read* syscall. Finally, we jump to this payload and execute it using a final ROP-gadget to open a shell. An in-depth discussion of the exploit would go beyond the scope of this paper and exceed the space available to present our research. Therefore, we made it available online[5] together with five other proof-of-concept exploits. Furthermore, we published an in-depth write-up about exploiting *unsafe* issues in Go as a series of blog articles[6].

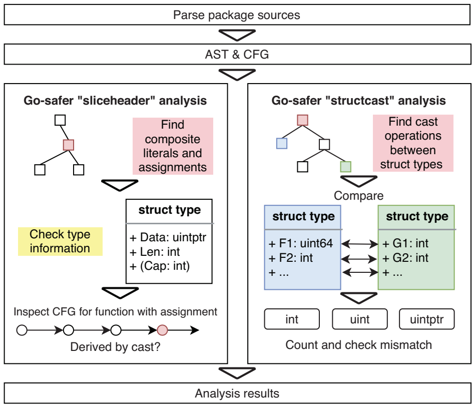

Figure 3: Architecture of *go-safer* static code analysis tool

3.2. *go-safer*: Finding Potentially Insecure Usages

We designed *go-safer*[7] to automatically give advice for some of the *unsafe* usage patterns introduced in the previous section. It is meant for assistance during manual audits and also for integration in build chains during development. Avoiding the patterns that *go-safer* detects prevents the garbage collector race and escape analysis flaw vulnerabilities that we discussed in Section 3.1. They are not covered by existing linters such as *go vet*. We found instances of these *unsafe* code patterns through the usage of *go-safer* in real-world code (cf. Section 4).

Figure 3 shows an overview of the architecture of *go-safer*. First, it uses *go vet* to build a list of packages to be analyzed and parses their sources. Then, a number of static code analyzers, called *passes*, run. Our analyses depend on existing passes to acquire the abstract syntax tree (AST) and control flow graph (CFG). Two separate analyses are run by *go-safer*: the *sliceheader* and the *structcast* passes.

The *sliceheader* pass discovers incorrect string and slice casts as shown in Listing 3. It finds composite literals

[5] https://github.com/jlauinger/go-unsafepointer-poc

[6] https://dev.to/jlauinger/exploitation-exercise-with-unsafe-pointer-in-go-information-leak-part-1-1kga

[7] https://github.com/jlauinger/go-safer

and assignments in the AST. Then, for each it checks whether the type of the receiver is *reflect.StringHeader*, *reflect.SliceHeader*, or some derived type with the same signature. For assignments, the analysis pass then finds the last node in the CFG where the receiver object's value is defined, and checks if it is derived correctly by casting a string/slice. If *go-safer* can not infer with certainty that the assignment receiver object was created by a cast, a warning is issued.

The *structcast* pass discovers instances of in-place casts between different struct types that include architecture-dependent field sizes. This can create a security risk when ported to other platforms because *unsafe* casts can lead to misaligned fields, and thus, memory access outside a value's bounds on some platforms, allowing the same exploit vectors as a buffer overflow does. The pass finds struct cast instances that involve *unsafe.Pointer* in the AST. Then, it compares the struct types and checks if they contain an unequal amount of fields with types *int*, *uint*, or *uintptr*, which are the architecture-dependent types supported by Go. If the numbers do not match, *go-safer* issues a warning.

4. A Study of Go's *unsafe* Usages in the Wild

We designed and performed a study of Go *unsafe* usage to answer the following research questions:

RQ1 How prevalent is *unsafe* in Go projects?
RQ2 How deep are *unsafe* code packages buried in the dependency tree?
RQ3 Which *unsafe* keywords are used most?
RQ4 Which *unsafe* operations are used in practice, and for what purpose?

In the following, we first describe our evaluation data set and then provide in-depth analyses of *unsafe* usage in the wild using *go-geiger*. Our evaluation scripts as well as the results are available online[8].

4.1. Data Set

As our research is focused on open-source projects, we crawled the 500 most-starred Go projects available on GitHub. To further understand the influence of dependencies, we then selected the applications supporting *go modules*. With the introduction of Go 1.13, *go modules*[9] are the official way to include dependencies. Unfortunately, 150 of the projects did not yet support Go modules. Thus, we excluded them from our set. Furthermore, 7 projects that did not compile were also removed. As a result, we ended up with 343 top-rated Go projects. These have between 72,988 and 3,075 stars, with an average of 7,860.

4.2. Unsafe Usages in Projects and Dependencies

We used the Go tool chain to identify the root module of each project. This is the module defined by the top-level *go.mod* file in the project. Then we enumerated the dependencies of the project, and built the dependency tree. For each package, we used *go-geiger* to generate CSV reports of the found *unsafe* usages. Through these analyses we answer the research questions of how many projects use *unsafe* either in their own code or dependencies (RQ1), and how deep in the dependency tree are the most *unsafe* code usages (RQ2). By selecting only results from the project root modules, we can easily find out how many applications contain a first-hand use of *unsafe* code. Our data shows that 131 (38.19%) projects have at least one *unsafe* usage within the project code itself. By looking closer at the imported packages, we see that 3,388 of 62,025 (5.46%) transitively imported packages use *unsafe*. There are 312 (90.96%) projects that have at least one non-standard-library dependency with *unsafe* usages somewhere in their dependency tree. Since all projects include the Go runtime, which uses *unsafe*, counting it as an *unsafe* dependency would mean that 100% of projects transitively include *unsafe*. We consider this to be less meaningful, as we assume the Go standard library is well audited and safer to use.

> Answer to RQ1
>
> About 38% of projects directly contain *unsafe* usages. Furthermore, about 91% of projects transitively import at least one dependency that contains *unsafe*.

Figure 4 shows the number of packages with at least one *unsafe* usage by their depth in the dependency tree for every project on its own as a heatmap, alongside the distribution for all projects combined as bars on the left side. It is evident that most packages with *unsafe* are imported early in the dependency tree with an average depth of 3.08 and a standard deviation of 1.62. This number is very similar to the overall average depth of imported packages (3.04). While the packages containing *unsafe* can be manually found and evaluated, this process requires significant resources to handle the increasing number of packages introduced through each dependency. For developers only the first level of dependencies, the ones they added themselves, are really obvious. On this level, 569 out of 8,952 imported packages (3.63%) contain *unsafe*.

> Answer to RQ2
>
> Most imported packages containing *unsafe* usages are found around a depth of 3 in the dependency tree.

4.3. Types and Purpose of Unsafe in Practice

This section answers RQ3 and RQ4. Figure 5 shows the distribution of the different *unsafe* types in our data set. Packages that are imported in different versions by the projects are counted once per version, as they might contain different *unsafe* usages and coexist in the wild. In our data set *uintptr* and *unsafe.Pointer* are used about equally often and are by far the most common with almost 100,000

[8] https://github.com/stg-tud/unsafe_go_study_results
[9] https://blog.golang.org/using-go-modules

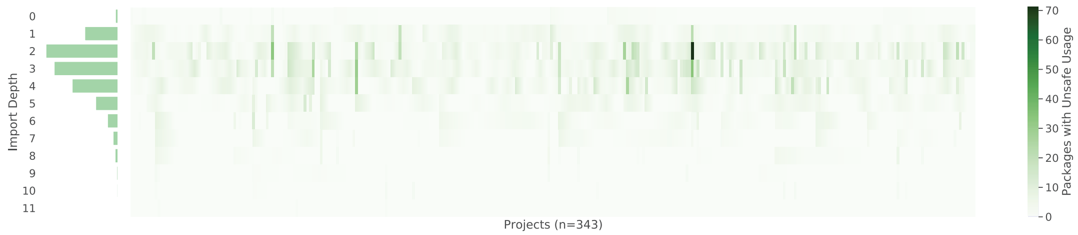

Figure 4: Import Depth of Unsafe Packages. Unsafe packages are around a depth of 3.08 (sd=1.62)

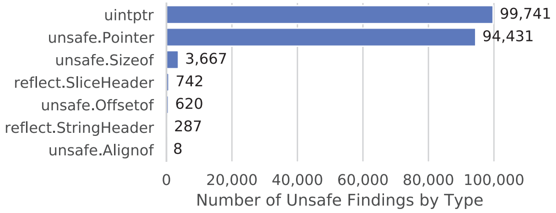

Figure 5: Distribution of different types of *unsafe* tokens

TABLE 1: Projects selected for labeled data set

	Name	Stars	Forks	Revision
1	kubernetes/kubernetes	66,512	23,806	`fb9e1946b0`
2	elastic/beats	8,852	3,207	`df6f2169c5`
3	gorgonia/gorgonia	3,373	301	`5fb5944d4a`
4	weaveworks/scope	4,354	554	`bf90d56f0c`
5	mattermost/mattermost-server	18,277	4,157	`e83cc7357c`
6	rancher/rancher	14,344	1,758	`56a464049e`
7	cilium/cilium	5,501	626	`9b0ae85b5f`
8	rook/rook	7,208	1,472	`ff90fa7098`
9	containers/libpod	4,549	539	`e8818ced80`
10	xo/usql	5,871	195	`bdff722f7b`

findings. Next, *unsafe.Sizeof* is still used a bit ($\sim 3,700$), while the other *unsafe* types are rarely used ($< 1,000$).

Answer to RQ3

In the wild, *uintptr* and *unsafe.Pointer* are orders of magnitude more common than other *unsafe* usages.

To learn about the purpose and context in which *unsafe* is used, we needed to manually analyze code. Thus, we selected the top 10 projects (Table 1) with the most *unsafe* usages in non-standard library packages. From these projects and all their transitive dependencies, we randomly sampled 400 code snippets that were found in the *standard library (std)* and 1,000 snippets from the remaining packages (*app*). We define standard library code as all packages that are part of the Go standard library or the *golang.org/x/sys* module, as the *syscall* standard library package is deprecated in favor of this module[10]. We split the snippets into two groups to analyze if there is a difference between the official standard library and non-standard library code regarding the usage of *unsafe*. Then, we identify class labels in two dimensions: what is being done, and for what purpose. Finally, we manually analyze all 1,400 code snippets and label them accordingly. The results of this process are shown in Table 2.

In the following, we outline the identified usage type classes describing what is being done in code. The most prevalent are *cast* operations from arbitrary types to other structs, basic Go types such as integers, slice/string headers, byte slices, or raw *unsafe.Pointer* values. The *memory-access* class is applied where *unsafe.Pointer* values are dereferenced, used to manipulate corresponding memory or for comparison with another address. *Pointer-arithmetic* denotes usages of *unsafe* to do some form of arithmetic manipulation of addresses, such as advancing an array. *Definition* groups usages where a field or method of type *unsafe.Pointer* is declared for later usage. *Delegate* are instances where *unsafe* is only needed in a function to pass it along to another function requiring a parameter of type *unsafe.Pointer*. Thus, the need to use *unsafe* is actually located elsewhere. *Syscall* are calls using the Go *syscall* package or *golang.org/x/sys* module. As the name suggests, *unused* is a class of occurrences that are not actually used in the analyzed code, e.g., dead code or unused parameters.

Our identified purpose classes, providing hints on why *unsafe* is used, are described in the following. *Efficiency* includes cases where *unsafe* is used only for the aim to improve time or space efficiency of the code. The *serialization* class contains (un)marshalling and (de)serialization operations such as in-place casts from complex types to bytes. *Generics* applies when *unsafe* is used to build functionality that would otherwise be solved with generics if they were available in Go. Samples in the *avoid garbage collection* class are used to tell the Go compiler to not free a value while it is used, e.g., by a function written in assembly. The *atomic operations* class contains usages of the *atomic* API which expects *unsafe* for some functions. The *foreign function interface (FFI)* class contains interoperability with C code (CGo), and calling functions that expect their parameters as *unsafe* pointers. *Hide from escape analysis* includes the pattern described earlier (Listing 2) to break the escape analysis chain. The *memory layout control* class contains code used for low-level memory management. *Types* snippets are used by the standard library to implement the Go type system. *Reflect* includes instances of type reflection and re-implementations of some types of the *reflect* package, e.g., using *unsafe.Pointer* instead of *uintptr* for slice headers. Again, *unused* is a class of unused occurrences.

Using *unsafe* for the sake of efficiency is the most prevalent motivation to use *unsafe* in the wild covering 58% in application code, whereas it is only used for this purpose in 5% of the cases in std. From these, 97% resp. 80% are achieved by casting different types. The second biggest reason to use *unsafe* in app is to perform some form

[10] https://golang.org/pkg/syscall

TABLE 2: Labeled unsafe.Pointer usages in application code (non standard library) and standard library samples
eff: efficiency, ser: (de)serialization, gen: generics, no GC: avoid garbage collection, atomic: atomic operations, FFI: foreign function interface, HE: hide from escape analysis, layout: memory layout control, types: Go type system, reflect: type reflection, unused: declared but unused

	eff		**ser**		**gen**		**no GC**		**atomic**		**FFI**		**HE**		**layout**		**types**		**reflect**		**unused**		**total**	
	app	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**	**app**	**std**
cast	562	16	178	33	18						24	6		2	3	13		45	1				786	115
memory-access	2	1	9									1			4	6		4					15	12
pointer-arithmetic	7	2	6	1						1		3	1	2	3	8		9					17	26
definition	4	1	23		2						4	5				9		8	6	3			39	26
delegate	4		64		2				11	5	29	45		4		14		6		1			110	75
syscall							17	138															17	138
unused																					16	8	16	8
total	579	20	280	34	22	0	17	138	11	6	57	60	1	8	10	50	0	72	7	4	16	8	1000	400

of (de)serialization, accounting for 28%. For the standard library, the most relevant motivation is avoiding garbage collection with 35%, whereas this is only used in 2% of the usages in the app sample. Furthermore, in std type 18%, FFI 15% and memory layout 13% related *unsafe* usages are rather common. Both subsets share that hiding from escape analysis with 0.1% (*app*) and 2% (*std*) and using *unsafe* for reflection with 1% (both) are rare. Implementation of generics functionality which is currently missing in Go is only done in few samples (2%), although some of the findings in the serialization class could alternatively be achieved with generics as well.

Answer to RQ4

More than half of the *unsafe* usages in projects and 3rd party libraries are to improve efficiency via *unsafe* casts. In the Go standard library, every third use of *unsafe* is to avoid garbage collection.

4.4. Vulnerable Usages

Looking at the study results, we see that *unsafe* is used consistently and wide-spread in the most popular open-source Go projects. One might argue that the usages found by *go-geiger* are only minor annoyances, not severe or would require a manual case-to-case inspection. Still, the exploitability of several of these usages was discussed in Section 3.1 with a reference to six proof-of-concept exploits that we developed. This clearly shows that it is indeed possible to use the memory corruptions to one's advantage. However, not all *unsafe* usages contain a vulnerability. As already discussed, we implemented more specific checks for two patterns known to be problematic in *go-safer* (Section 3.2). With it, three of the proof-of-concept exploits are mitigated, leaving the others which are much harder to detect statically for future work. The application of *go-safer* to our data set revealed more than 60 insecure usages of *unsafe* in different projects. Based on the results, we submitted so far 14 pull requests to fix these usages. By now, 10 have already been reviewed, acknowledged, and accepted by the corresponding project maintainers.

5. Threats to Validity

Potential internal threats to validity for our study include bias towards bigger projects because those might be over-represented in the manually labeled data set. External threats include a bias towards more active projects with many developers because we selected a subset of the most-starred open-source projects on GitHub. Also we only considered projects that use the Go module system and about a third of the top 500 projects are not covered by the analysis yet. Further, we could have missed projects from a special domain not having that many stars which might have other usage scenarios for *unsafe* Go. Nevertheless, one can argue that the biggest projects also have professional developers, higher standards and code gets more reviewed, thus, code quality should be higher.

6. Related Work

Previous research on Go mostly concentrated on issues related to its concurrency model including the channel implementation [6], [7], [8], [9], [10], [11]. The work by Wang et al. [4] suggests an improvement of the existing escape analysis in Go which we also discussed in our paper.

Moreover, the usage of *unsafe* in other languages has already been studied to varying degrees. For Java, Mastrangelo et al. [12] identified that 25% of the analyzed artefacts depend on the Java *unsafe* library. The different JVM crash patterns caused by those usages are analyzed by Huang et al. [13]. Recently, two studies analyzed *unsafe* usages in Rust projects and identified that *unsafe* is widely used to improve performance or to reuse existing code [3], [14]. Furthermore, work was presented on how to ensure memory safety while using *unsafe* in Rust [15]. Lehmann et al. [16] studied to which extent *unsafe* programs compiled to WebAssembly can lead to vulnerabilities within the virtual machine environment. For C/C++, non memory-safe languages, research exists on how to support at least partial memory safety [17], [18] and work on identifying vulnerabilities by program analyses [19]. A comprehensive study on memory-management-related vulnerabilities, like the ones we discussed earlier, and their mitigations is presented in earlier work [20].

Concerning project dependencies, it is difficult to count the dependencies that matter the most, e.g., by excluding test dependencies [21]. A common problem is that dependencies are often updated slowly, keeping old bugs alive, although measures such as automated pull requests exist to mitigate this problem [22], [23], [24].

7. Conclusion

In this paper, we gave a systematic description of different dangerous programming patterns involving *unsafe* and novel evidence on how to exploit these patterns. Furthermore, we presented two novel tools to help Go developers write safer code with respect to *unsafe* Go and security analysts to evaluate *unsafe* code. First, *go-geiger* identifies *unsafe* usages not only within the main project package, but also in its transitive dependencies. Therefore it represents an effective tool to focus audit efforts on the code locations that are the most dangerous, raising awareness into how *unsafe* is included into a project, and helps getting a general sense for the potential risks of deploying a specific project. Second, *go-safer* is a new static code analysis tool that helps developers identify dangerous code patterns that were previously uncaught with existing tools for linting. Additionally, we conducted a study of 62,025 packages from 343 top-starred open-source Go projects. Here, we have shown that *unsafe* is very common, especially when taking project dependencies into account. Finally, derived from this study, we presented a new data set of manually labeled code snippets, providing insight into how and for what purpose *unsafe* is used by developers. The reasons for introducing *unsafe* operations are often tied to optimization, interoperability with external libraries or to circumvent language limitations.

In the future, supervised learning algorithms could use our labeled data set to train classifiers, which can then identify the purpose and domain of *unsafe* usages by looking at new code. Furthermore, plugins for common IDEs that integrate our tools, *go-geiger* and *go-safer*, could be built to incorporate them into developers' workflow.

Acknowledgments

This research work has been funded by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] S. M. Alnaeli, M. Sarnowski, S. Aman, A. Abdelgawad, and K. Yelamarthi, "Source Code Vulnerabilities in IoT Software Systems," *Advances in Science, Technology and Engineering Systems Journal (ASTES)*, vol. 2, no. 3, pp. 1502–1507, Aug. 2017.

[2] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, June 2001, p. 177–190.

[3] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust Used Safely by Software Developers?" in *IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, May 2020.

[4] C. Wang, M. Zhang, Y. Jiang, H. Zhang, Z. Xing, and M. Gu, "Escape from Escape Analysis of Golang," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, May 2020.

[5] A. Sibiryov, "Golangs garbage," in *SREcon17*. USENIX Association, May 2017.

[6] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding Real-World Concurrency Bugs in Go," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Apr. 2019, pp. 865–878.

[7] N. Dilley and J. Lange, "An Empirical Study of Messaging Passing Concurrency in Go Projects," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Feb. 2019, pp. 377–387.

[8] M. Giunti, "GoPi: Compiling linear and static channels in go," in *Coordination Models and Languages (COORDINATION)*. Springer International Publishing, Jun. 2020, pp. 137–152.

[9] J. Gabet and N. Yoshida, "Static Race Detection and Mutex Safety and Liveness for Go Programs," in *European Conference on Object-Oriented Programming (ECOOP)*, Nov 2020.

[10] J. Lange, N. Ng, B. Toninho, and N. Yoshida, "Fencing off go: Liveness and safety for channel-based programming," in *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2017, p. 748–761.

[11] E. Bodden, K. I. Pun, M. Steffen, V. Stolz, and A.-K. Wickert, "Information flow analysis for go," in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA)*. Springer International Publishing, Oct. 2016, pp. 431–445.

[12] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: The java unsafe api in the wild," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 2015, p. 695–710.

[13] S. Huang, J. Guo, S. Li, X. Li, Y. Qi, K. Chow, and J. Huang, "SafeCheck: Safety Enhancement of Java Unsafe API," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, May 2019, pp. 889–899.

[14] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Jun. 2020, pp. 763–779.

[15] H. M. J. Almohri and D. Evans, "Fidelius charm: Isolating unsafe rust code," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, Mar. 2018, p. 248–255.

[16] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *USENIX Security Symposium (USENIX Security)*. USENIX Association, Aug. 2020.

[17] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CUP: Comprehensive User-Space Protection for C/C++," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, May 2018, p. 381–392.

[18] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Jun. 2009, p. 245–258.

[19] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: sanitizing for security," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 1275–1295.

[20] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2013, pp. 48–62.

[21] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable Open Source Dependencies: Counting Those That Matter," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Oct. 2018, pp. 1–10.

[22] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Oct. 2017, p. 2187–2200.

[23] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Oct. 2017, pp. 84–94.

[24] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web," *Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.