Identification and Analysis of unsafe.Pointer Usage Patterns in Open-Source Go Code

It's dangerous to Go alone. Take *this!

Master thesis by Johannes Tobias Lauinger
Date of submission: October 26, 2020

1. Review: Prof. Dr.-Ing. Mira Mezini
2. Review: Anna-Katharina Wickert, M.Sc.
3. Review: Dr. rer. nat. Lars Baumgärtner
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
Department
Software Technology Group

To fritz, without which none of this
would have happened.

Abstract

One decade after its first published version, the Go programming language has become a popular and widely-used modern programming language. It aims to achieve thorough memory and thread safety by using measures such as a strict type system and automated memory management with garbage collection, which prevents invalid memory access. However, there is also the *unsafe* package, which allows developers to deliberately circumvent this safety net. There are a number of legitimate use cases for doing this, for example, in-place type conversions saving reallocation costs to improve efficiency, or interacting with C code through the foreign function interface.

However, misusing the *unsafe* API can lead to security vulnerabilities such as buffer overflow and *use-after-free* bugs. This work contributes an analysis of *unsafe* usage patterns with respect to a security context. It reveals possible code injection and information leak vulnerabilities in proof-of-concept exploits as well as common usages from real-world code.

To assess the risk of *unsafe* code in applications, this work presents *go-geiger*, a novel tool to help developers quantify *unsafe* usages not only in a project itself, but including its dependencies. Using *go-geiger*, a study on *unsafe* usage in the top 500 most popular open-source Go projects on *GitHub* is conducted, including a manual study of 1,400 individual code samples on how *unsafe* is used and for what purpose. The study shows that 5.5% of packages imported by the projects using the Go module system use *unsafe*. Furthermore, 38.19% of the projects use *unsafe* directly, and 90.96% include *unsafe* usages through any of their dependencies. A replication and comparison of a concurrent study by Costa et al. [10] matches these results.

This work further presents *go-safer*, a novel static code analysis tool that helps developers to identify two dangerous and common misuses of the *unsafe* API, which were previously undetected with existing tools. Using *go-safer*, 64 bugs in real-world code were identified and patches have been submitted to and accepted by the maintainers. An evaluation of the tool shows 95.5% accuracy on the data set of labeled *unsafe* usages, and 99% accuracy on a set of manually inspected open-source Go packages.

Zusammenfassung

Ein Jahrzehnt nach der ersten veröffentlichten Version ist die Programmiersprache Go heute eine beliebte und weit verbreitete, moderne Sprache. Sie strebt Speicher- und Threadsicherheit durch Maßnahmen wie ein striktes Typsystem und automatische Speicherverwaltung, die ungültige Speicherzugriffe verhindert, an. Es gibt allerdings ebenfalls das *unsafe* Package, eine API, die es Entwickler*innen erlaubt, diese Maßnahmen zu umgehen. In manchen Fällen kann dies gerechtfertigt sein, beispielsweise bei der Konvertierung von Daten in einen anderen Typ, ohne diese im Speicher zu kopieren, um so die Effizienz des Programms zu steigern, oder um externen C Code über das Foreign Function Interface zu nutzen.

Eine falsche Benutzung der *unsafe* API kann jedoch zu Sicherheitsproblemen wie Buffer Overflows und *Use-After-Frees* führen. Diese Arbeit analysiert Verwendungsmuster von *unsafe* Code im Hinblick auf Sicherheitsrisiken. Dabei werden mögliche Code Injection und Information Leak Verwundbarkeiten sowohl in Proof-of-Concepts als auch in realem Anwendungscode zu Tage gebracht.

Um die Risiken durch *unsafe* Code in Anwendungen abzuschätzen, stellt diese Arbeit *go-geiger* vor. Es handelt sich dabei um ein neues Werkzeug, das Entwickler*innen dabei hilft, *unsafe* Nutzungen in Projekten und deren Abhängigkeiten zu finden. Mit *go-geiger* wird eine Studie zur Nutzung von *unsafe* in den 500 beliebtesten Open-Source Go Projekten auf *GitHub* durchgeführt, inklusive einer manuellen Analyse von 1,400 individuellen Codestücken in Bezug darauf wie und zu welchem Zweck *unsafe* benutzt wird. Die Studie zeigt, dass 5.5% der Packages, die von Projekten importiert werden, welche das Go Modules System unterstützen, *unsafe* verwenden. Darüber hinaus nutzen 38.19% der Projekte *unsafe* direkt, und 90.96% enthalten *unsafe* Code durch ihre Abhängigkeiten. Eine Replikation sowie ein Vergleich mit einer zeitgleichen Studie von Costa et al. [10] bestätigt diese Ergebnisse.

Weiterhin präsentiert diese Arbeit *go-safer*, ein neues statisches Analysewerkzeug, das Entwickler*innen hilft, zwei gefährliche und häufig vorkommende inkorrekte Verwendungen der *unsafe* API, die mit bisher existierenden Tools nicht gefunden werden, zu identifizieren. Mittels *go-safer* konnten 64 Fehler in realem Code gefunden und entsprechende Patches eingereicht werden, die von den Maintainern bestätigt wurden. Eine Evaluation des Tool ergibt eine Accuracy von 95.5% auf dem Datensatz von *unsafe* Codezeilen, und 99% Genauigkeit auf händisch analysierten Open-Source Go Packages.

Acknowledgments

This thesis was written at Technische Universität Darmstadt under the supervision of Prof. Dr.-Ing. Mira Mezini, Anna-Katharina Wickert, M.Sc, and Dr. rer. nat. Lars Baumgärtner. Parts of this work have been submitted as a research paper titled "Uncovering the Hidden Dangers: Finding Unsafe Go Code in the Wild", which is accepted at the 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom 2020) [33]. I explicitly acknowledge the contributions of the co-authors Lars Baumgärtner, Anna-Katharina Wickert, and Mira Mezini. I made sure to properly cite all relevant prior and concurrent work as usual with the standards on scientific ethics to clearly show which contributions are derived from the work of other people and which are mine.

I would like to thank Lars, Anna, and Mira for their excellent work supervising this thesis. I could not imagine having had any better support during the past months. Our regular meetings have had priceless value for the success of this work. Especially, I want to thank Anna for restoring my motivation in challenging times, and Lars for his honest, demanding, and always very helpful feedback.

In addition, I want to thank my family for giving me both moral and financial support throughout my entire studies, always believing in me and having my back, and my girlfriend for cheering me up even in the most stressful and hard days. In fact especially in those days. Finally, a big thank you to my friends who have made the past years at university the great time that it has been, Timo for letting me switch offices for a couple of days, and Anna, John, Tim, Tobias, and Yvonne.

Contents

1. Introduction

In today's modern society, software has become an integral part of many industries and areas of life. The security of that software is very important, because successful attacks on it can have serious implications, especially in the case of critical infrastructures like energy and food supply chains, or medical services. It is therefore important to try reducing the attack surface in the best possible way. In the last decade, there has been an ongoing adoption of memory-safe languages for many different applications. Such languages include, for example, Google's Go, Rust, Nim, and even Java. Memory safety is one of the most important areas of software security, because a large number of vulnerabilities are caused by memory access bugs [23]. In fact, Microsoft, for example, reports that memory safety accounts for around 70% of all their bugs[1]. To reduce the risk of such vulnerabilities, memory-safe languages provide different mechanisms to protect potentially dangerous operations, such as accessing raw memory, dereferencing raw pointers, or arbitrary conversions between incompatible types. However, these languages often also provide ways for developers to explicitly circumvent the safety measures to various degrees.

This thesis is focused on the Go programming language. Go uses a strict type system with limiting rules on pointer access and cast operations, and automatic memory management, to achieve a high level of memory and thread safety [46]. However, it also offers the *unsafe* package. This package is an API built into the language that allows arbitrary access to raw pointers, similar to the way pointers in C are handled. There are legitimate use cases for this, such as in an application with time and memory constraints that needs to cast values to different types without reallocating them, or to access hardware when building a driver.

Thesis contribution big picture

The high-level objective of this thesis is to explore the Go *unsafe* API, both by finding possible security vulnerabilities and by analyzing how it is used in applications.

It is dangerous to use the *unsafe* API because, when used incorrectly, it can cause memory safety bugs that lead to exploitable security vulnerabilities, as is shown in this thesis. There can be buffer overflows leading to possible code injections when incompatible types with different sizes or memory alignments are converted into each other. Also, the compiler may be unable to correctly

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

determine the lifetime of a value and allocate it on a function stack instead of the program heap, which can lead to *use-after-free* vulnerabilities, and with them all kinds of malicious program behaviors. Thus, when *unsafe* code is used it must be audited by the developers at least.

Checking *unsafe* code in a project can be hard because it can be introduced not only through first-party code, but also through dependencies. For Rust programs, which contain an *unsafe* feature similar to Go's, Evans et al. [16] recently showed that *unsafe* blocks are often introduced through third-party libraries. It might not be directly obvious which dependencies contain *unsafe* code and should be audited, and checking all of them is tedious and would create a tremendous cost in terms of development time. Therefore, security analysts, software developers, and system administrators need tools that support them in identifying *unsafe* code usages in their project, including its dependencies, and assessing their risk. There are suitable tools for other languages, like *cargo geiger* for Rust code, but previously there was no such tool for Go.

The goal of this thesis is to examine the dangers that can come from the use of the *unsafe* API in Go code, build a tool similar to *cargo geiger* for Go developers, and to find out how and to which extent *unsafe* is used in actual projects.

1.1. Problem Statement

This section describes how the thesis objective is split into different parts that are connected together. Figure 1.1 illustrates the organization and high-level contributions of this thesis.

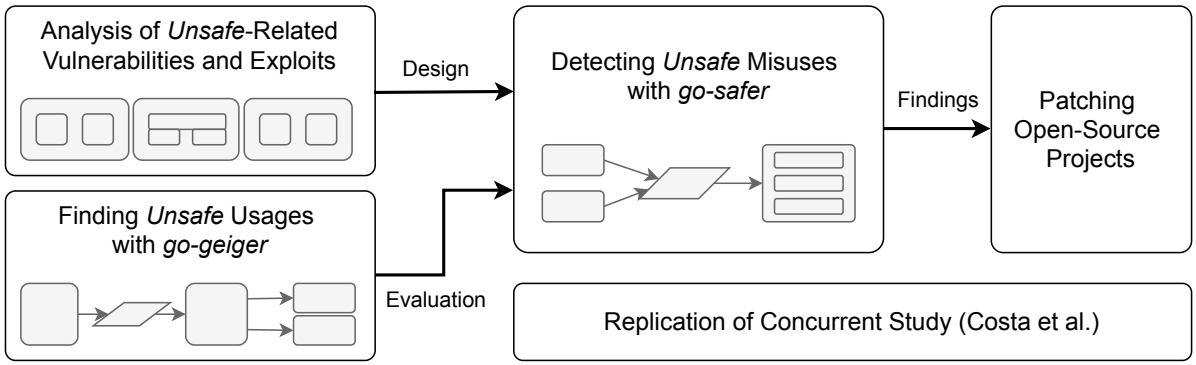


Figure 1.1.: Problem statement and thesis organization.
Parts in gray outline chapters and are shown in detail there.

First, a thorough manual analysis of possible vulnerabilities that can result from incorrect usages of the *unsafe* API, including their consequences, is performed. This is shown in the top left corner in Figure 1.1. Then, *go-geiger* is designed. It is a novel tool that finds *unsafe* usages in Go source code, including its dependencies. This tool is used for an empirical study on the usage

of *unsafe* in the wild. First, open-source Go projects are downloaded from *GitHub*. They are analyzed using *go-geiger*, which yields raw data about *unsafe* usages. This data is evaluated both quantitatively in terms of a statistical analysis, and qualitatively by sampling, manually reviewing, and labeling code snippets sampled from the identified *unsafe* usages. With these labels, insights are found about how and for what purpose the *unsafe* API is used. This study, as well as *go-geiger*, is shown in the bottom left corner in Figure 1.1. To contribute to safer usage of *unsafe*, a novel linter called *go-safer* is developed. It is illustrated in the top center. Its design is based on the *unsafe*-related vulnerabilities and *unsafe* usage patterns in open-source projects, and it is evaluated using identified usage data about *unsafe*. Next, findings of *go-safer* are reported to the respective project maintainers, and patches are sent to fix them. Figure 1.1 indicates this in the top right corner. Finally, as shown in the bottom center, a concurrent study on *unsafe* in Go [10] is replicated and compared to the results of this work. Boxes that contain further structure inside in gray are shown in more detail in a figure at the beginning of their chapters.

Thus, this thesis mainly presents an analysis of vulnerabilities that are caused by misuses of the *unsafe* API, as well as two novel static analysis tools for Go developers. It is worth noting that both tools have their own design, implementation, and evaluation sections in their respective chapters.

1.2. Contributions

The main contributions that are made in this thesis are the following:

1. A thorough analysis of problems and consequences of usage patterns of the *unsafe* API in Go code with respect to a security context, revealing three main areas of danger,
2. *go-geiger*, a novel open-source static analysis tool to identify and count *unsafe* usages in Go packages, including their dependencies,
3. a quantitative study of *unsafe* code usage in 343 of the top 500 most popular open-source Go projects on *GitHub*,
4. an in-depth study of 1,400 code samples used in ten selected projects, yielding a data set of two-dimensional manual classifications of usages and valuable insight into how and for what purpose *unsafe* code is used in Go applications,
5. *go-safer*, a novel open-source, *go vet*-style, linter tool to find two dangerous and common *unsafe* usage patterns that were previously uncaught with existing tools, including an evaluation of its performance,
6. the submission of 14 pull requests to project maintainers with fixes to 64 previously vulnerable code snippets in open-source Go libraries, and

7. a replication of a related study on *unsafe* Go code in concurrent work by Costa et al. [10], including a comparison to this work and discussion of differences.

Parts of these contributions have been published in [33]. For Chapter 3, Sections 3.2 and 3.3.3 are published in the paper. With respect to *go-geiger*, Sections 4.3.3, 4.3.4, and 4.3.5 are presented for the first time in this thesis. The other sections in Chapter 4 are discussed in the paper, although with less detail. In Chapter 5, the evaluation of *go-safer* in Section 5.3 is novel in this thesis. The content of the remaining chapters is new in this work, no significant and substantial parts of those have been published in the paper. Furthermore, a series of blog posts[2] covers parts of the *unsafe*-related exploits presented in Chapter 3.

1.3. Outline

The remainder of this thesis is structured as described in the following. Chapter 2 gives background information on the Go *unsafe* API and other concepts needed for this thesis. Chapter 3 analyzes and discusses possible vulnerabilities caused by *unsafe* code usages. Then, Chapter 4 presents the design, implementation, and evaluation of *go-geiger*, the novel static analysis tool which finds and counts *unsafe* usages. The chapter also describes the empirical study on *unsafe* code in the wild, and presents the labeled data set of *unsafe* code samples. After that, Chapter 5 shows the design, implementation, and evaluation of *go-safer*, the novel, *unsafe*-focused linter tool. Next, Chapter 6 reviews related and concurrent work. Chapter 7 discusses the meaning and impact of the findings and results of this thesis. Finally, Chapter 8 concludes the work, and Chapter 9 outlines possible future research.

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

2. Background

This chapter presents background information that is necessary for this thesis. First, the *unsafe* API as well as slices and strings in Go are introduced. Then, Go memory management is discussed, and exploit techniques and mitigations are presented. After this, Go dependency management is covered, and the chapter concludes with concepts of static code analysis.

2.1. Go *Unsafe* API

The Go *unsafe* package[1] is an application programming interface (API) that allows developers to use raw pointers with all their benefits and dangers. As such, it offers an escape hatch to circumvent safety measures provided by Go. Listing 2.1 shows the functions and types included in the *unsafe* package.

Listing 2.1: Unsafe package API

```
1 package unsafe
2
3 type Pointer
4
5 func Alignof(x ArbitraryType) uintptr
6 func Offsetof(x ArbitraryType) uintptr
7 func Sizeof(x ArbitraryType) uintptr
```

There are three functions *Alignof*, *Offsetof*, and *Sizeof* (Lines 5–7), as well as the type *Pointer* (Line 3). The *unsafe.Pointer* is the most significant package member with respect to the possibilities that it offers. It is a pointer type that is exempt from the restrictions that normally apply in order to improve memory safety. In particular, *unsafe* pointers can be converted to and from any other pointer type, allowing arbitrary casts between any type. Furthermore, they can be converted to and from *uintptr* values, which is an integer type with sufficient size to store pointers on the specific target architecture of the program. Since *uintptr* values behave like normal integer variables, they can be used in arithmetic expressions. Therefore, *unsafe* pointers can also be used to do pointer arithmetic. With *unsafe.Pointer*, developers can achieve the same flexibility as with raw *void** pointers in C.

[1] `https://golang.org/pkg/unsafe`

Listing 2.2 shows examples of both *unsafe.Pointer* features. The *floatToBits* function (Lines 3–5) illustrates how the variable *x* of type *float32* is first cast to an *unsafe* pointer and then to a value of type *int* (Line 4). Since the *unsafe* conversion rules only allow arbitrary casts between any pointer type, *x* is first converted into a pointer value and the resulting *int* pointer is dereferenced before returning the final value. This conversion yields the raw bits of the floating point value stored using the IEEE 754 format [26]. It would be impossible using the normal Go conversion rules, because *float32* and *int* are incompatible types. The *secondByte* function (Lines 7–9) shows how pointer arithmetic is used to manually iterate over an array type. The function expects an array with a length of five bytes and returns the second byte. Instead of direct indexing, it achieves this by obtaining the address of the array as a *uintptr* value, adding 1 and then casting the resulting address to a *byte* value, again taking care of dereferencing the resulting pointer type (Line 8).

Listing 2.2: Usage examples of the Go unsafe API

```
1 import "unsafe"
2
3 func floatToBits(x float32) int {
4     return *(*int)(unsafe.Pointer(&x))
5 }
6
7 func secondByte(b [5]byte) byte {
8     return *(*byte)(unsafe.Pointer(uintptr(unsafe.Pointer(&b)) + 1))
9 }
```

The other functions of the *unsafe* packages are evaluated at compile time and can be used to gain knowledge about the alignment, offset, or size of a field in a structure type relative to the beginning of the structure. These can be necessary when dealing with the direct byte representations of structure types such as when implementing network protocols or device drivers. In this thesis, the term *unsafe* usage refers to all members of the *unsafe* package, as well as the slice and string header structures that are described in the next section.

2.2. Slices and Strings in Go

Slices in Go (e.g., the *[]byte* type) are views on arrays[2]. They are a data type consisting of an underlying memory area storing the array data, as well as information about length and capacity of the slice. The Go runtime takes care that there are no accesses outside the slice bounds. The length indicates how many elements are contained in the slice, and the capacity denotes the allocated space in memory. The capacity is always equal or greater than the length. If it is greater, elements can be added by increasing the length without allocating new memory. If the

[2] `https://blog.golang.org/slices`

length is equal to the capacity, and a new element is added, the Go runtime allocates a new, bigger underlying memory array and copies the existing data. When a subset of the slice is taken by specifying start and end indices, a new slice is created that uses the same underlying data array as the original one. It might have a capacity longer than its length if there are further elements in the original slice after the end of the newly-taken subslice.

Strings in Go are read-only *[]byte* slices[3]. Usually, they are encoded in UTF-8, but this is not required. The length of a string is always the number of bytes needed to store its encoded form, which is often not the same as the number of characters in the string. Strings are immutable, because string literals are placed in a constant data section of the binary when the code is compiled. This means that they are usually located within a read-only memory page at runtime, and mutating them would cause the program to receive a segmentation fault signal and crash. Since strings can not be changed, there is no need to have a capacity that is different from the length, therefore there is no capacity information available for strings.

It is possible to access the internal structure of slices and strings by casting them to a slice or string header representation using the *unsafe* API. Listing 2.3 shows the API of these header types, which are part of the Go *reflect* package[4].

Listing 2.3: Reflect slice and string header types

```
1  package reflect
2
3  type SliceHeader struct {
4     Data uintptr
5     Len int
6     Cap int
7  }
8
9  type StringHeader struct {
10    Data uintptr
11    Len int
12 }
```

As the name suggests, slices correspond to the *SliceHeader* type (Lines 3–7), while strings are associated with the *StringHeader* type (Lines 9–12). The *Len* fields (Lines 5 and 11) denote the current length of the slice or string, and the *SliceHeader* type has an additional *Cap* field (Line 6) for the slice capacity. The reference to the underlying data array is a *uintptr* value in both types (*Data* in Lines 4 and 10). This means that it is not inherently a pointer type, instead it stores the plain address of the array. However, the Go runtime recognizes slices and strings and, as a special case, treats their *Data* field like a pointer.

Slices are different from array or buffer types like *[10]byte*, which have a fixed length. There is no way to grow an array by appending more data than it can store. Furthermore, the Go type

[3] https://blog.golang.org/strings
[4] https://golang.org/pkg/reflect

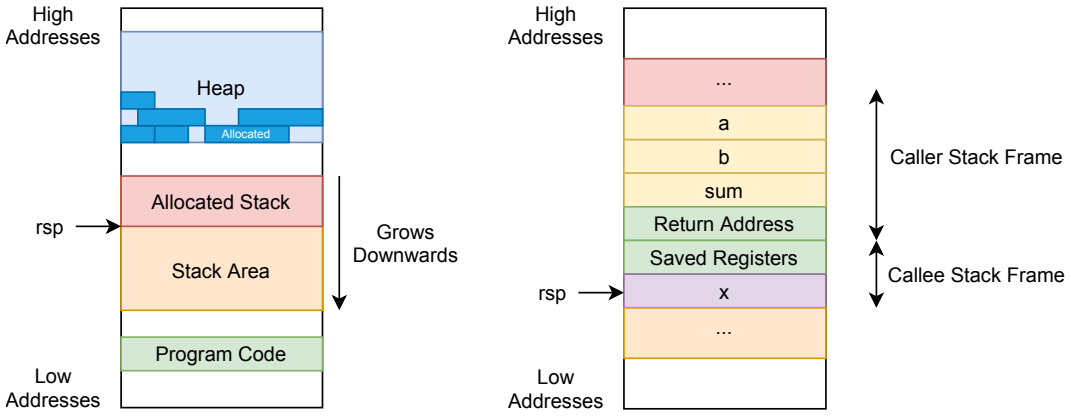
system makes the array length a fundamental property of the type, which means that there can not be an implicit conversion between arrays of different lengths. In contrast, slices store their length in a variable and are not treated differently by the type system depending on their length.

2.3. Memory Management in Go

The Go programming language uses automatic memory management, which means that developers do not need to manually allocate and free memory for data structures. Instead, a value is simply declared and the compiler adds the code to properly allocate the appropriate space in memory. This section first describes how the stack and heap are used in Go, and then introduces garbage collection and escape analysis.

2.3.1. Stack and Heap

The available memory addresses that an application can access are divided into logical sections. Go programs always at least use a program code, stack, and heap section, although there can be more. Figure 2.1a shows an illustration of the distribution of these in the address space.



(a) Process memory layout

(b) Stack frame layout for function add in Listing 2.4

Figure 2.1.: Go memory and stack frame layout

The program code (shown in green) is loaded at a relatively low address. Then, the stack area (red and orange) follows, and finally the heap section (shown in blue) is located at higher addresses. The stack area is used to store data that is associated with individual functions, and will be needed only as long as the functions run [17]. In contrast, the heap contains values with a lifetime that is longer than the execution of a specific function, for example, global variables.

In Go, developers do not actively decide whether a variable is allocated on the stack or heap, instead this is done by the compiler.

The stack pointer *rsp* (this name is specific to the *amd64* architecture, but the concept applies to other architectures as well) is a register that contains the address of the last value that was allocated on the stack. As shown in Figure 2.1a, it thus marks the border of the used part of the stack area. When new values are stored (pushed) on the stack, the stack pointer is decremented and the data is stored at the resulting new address in *rsp*. Therefore, the stack grows downwards, from high to low addresses.

Data on the stack is organized in frames. When a function is called, a new stack frame is created for it. Figure 2.1b shows a visualization of this. It presents the stack after calling the *add* function in Listing 2.4.

Listing 2.4: Example function to illustrate Go stack frames

```
1  func add(a, b int) (sum int) {
2      x := a + b
3      sum = x + 42
4      return sum
5  }
```

In Go, function parameters and return values are both passed entirely on the stack[5]. Therefore, the parameters *a* and *b*, as well as the return value *sum* (declared in Line 1) are pushed onto the stack by the caller function. They are shown in yellow in Figure 2.1b. The last value that gets pushed into the caller frame is the address of the next instruction to execute after the callee function returns. It is called return address or return instruction pointer (RIP) and shown in green. Then, the new stack frame for the callee begins with the CPU register contents at the function call time. These are stored on the stack, so that they can be restored to the original values before returning back to the caller. They are shown in green as well in Figure 2.1b. After this, local variables of the callee function that do not surpass its lifetime are pushed to the stack. In this case, there is the variable *x* (Line 2 in Listing 2.4), which is shown in purple in Figure 2.1b. The *sum* variable (Line 3) is not pushed to the callee stack frame, because, since it is a return value, it was already placed in the caller stack frame. When the function *add* returns, the stack pointer *rsp* will be incremented up to the *sum* value to destroy the callee stack frame, and the CPU will jump to the return address that was stored on the stack.

In contrast, data on the heap is organized in chunks [17]. There can be several allocated chunks of data on the heap, and they are not necessarily adjacent. Figure 2.1a shows used heap areas in a darker shade of blue. Adding data to the heap requires calling special allocation functions that are provided by the runtime, which in the case of Go is done automatically by the compiler. Furthermore, memory on the heap must be freed when it is no longer used, otherwise the program will eventually run out of usable heap space. Therefore, storing data on the heap has

[5] https://go.googlesource.com/proposal/+/master/design/27539-internal-abi.md

more overhead than storing it on the stack, because deallocation on the stack is done with a single addition when destroying an entire stack frame.

2.3.2. Garbage Collection (GC)

The Go runtime uses a concurrent move-and-sweep garbage collector (GC) to free unused heap memory [46]. It is triggered by one of two conditions, either when the heap usage has increased to a specific size (usually when it has doubled since the last GC run), or after a fixed time has passed without any GC runs. Figure 2.2 shows a visualization of the marking phase in a GC run. Rounded boxes represent variables, rectangular boxes are allocated values on the heap, and arrows indicate references from pointer types to their targets.

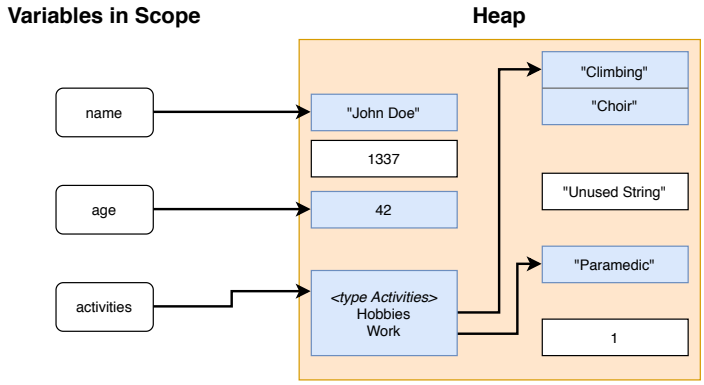


Figure 2.2.: Garbage collector mark phase visualization

The GC takes all variables that are present in the scope of the program and marks their values as live. These variables can be allocated either on the stack, which is not managed by the GC, or on the heap. In this case, there are the *name*, *age*, and *activities* variables with the values "*John Doe*", *42*, and *Activities*, which are shown in blue in Figure 2.2. Then, it recursively follows all pointer types within live values and marks their target values as live, too. This concerns the values "*Climbing*", "*Choir*", and "*Paramedic*". However, the values *1337*, "*Unused String*", and *1* are not reachable from any pointer variables in scope, and therefore are not marked. They are shown in white in the figure.

After this marking phase is completed, the sweep phase frees all memory that is not marked as live. Finally, the markings are reset and a new GC run can be triggered.

Using a GC improves memory safety in Go, because without explicit deallocations, the risk of dereferencing pointers that have already been freed (*use-after-free*) is reduced. The Go GC runs concurrently with all other threads of a program. During the mark phase, it briefly stops the execution of the other threads to ensure a consistent view on all live heap values. The sweep

phase is executed in parallel to the other threads.

2.3.3. Escape Analysis (EA)

Escape analysis (EA) is a technique to decide whether a variable can be placed on a function's stack frame, or needs to be allocated on the heap [8]. It is used by the Go compiler when a variable is declared. Go prefers to allocate on the stack if possible, because there is no explicit deallocation step needed to free the value there. Instead, the stack frame is removed entirely when the function returns.

In EA, a value is said to escape if references to it can live longer than the lifetime of the current function. If it does escape, it needs to be allocated on the heap so that when the function has returned, references that still exist can continue to access the value in memory even though the stack frame has vanished. Otherwise, it can be placed on the stack.

The EA algorithm employed by the Go compiler works by checking for each variable that is declared in a function whether a reference to it is stored in another value or passed to another function. If there is a reference stored in another object, this object has to be checked too, and if it escapes so does the original variable. The same is true for functions. If the variable gets passed to another function, the EA algorithm transitively looks into that function and determines if the value escapes there. If it does, it must also be treated as an escaped value in the original function. Finally, if the function returns a reference to the variable, it obviously escapes as well.

2.4. Exploit Techniques and Mitigations

This section briefly explains common exploit techniques (using a buffer overflow with return-oriented programming to redirect control flow), as well as common mitigations against them.

2.4.1. Buffer Overflow

A frequent security vulnerability related to memory safety is the buffer overflow [32]. It happens when a buffer variable, for example, a slice, is too short for the data that is written to or read from it, causing adjacent memory to be accessed and possibly overwritten. If information is retrieved by the attacker from variables that are located next to the buffer, a successful *information leak* exploit is achieved. Overflows can happen anywhere in the memory, however a common source of security problems are stack-based buffer overflows. When a buffer variable is placed on the stack and too much data gets written to it, the data will be written to higher memory addresses. After overwriting the other local variables and saved registers, this can cause the saved return address to get changed. If an attacker can control the data that is placed there, they can change

the program flow by carefully choosing the address that the CPU jumps to after the current function returns.

Figure 2.3 shows a visualization of an example of input data used to execute arbitrary code by overwriting the saved return address. This is called *stack smashing* [48]. First, arbitrary data is written to fill up the buffer and any adjacent local variables up to the saved return address. In this case, this data is composed of several *A* characters. Next, the desired new return address is written (RIP). This will replace the original, correct address. By using the address of the following data that gets written to the stack, the program control flow can be redirected to attacker-controlled code.

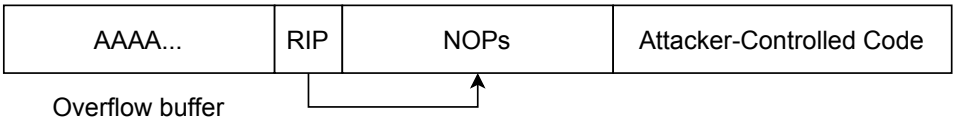


Figure 2.3.: Stack smashing payload to inject exploit code

Since stack addresses can be slightly unpredictable in practice, for example, because environment variables with different lengths might be located on the stack, a *NOP slide* can be used to make the exploit more robust. *NOP*, or *no-op*, is a machine instruction that does nothing at all, the processor continues with the following instruction. A *NOP slide* is a series of such instructions [52]. Using this, it is sufficient to overwrite the return instruction pointer with an address somewhere within the *NOP* instructions to make the processor go through all remaining of them until it reaches the exploit code, similar to sliding down a slope. Thus, the address does not have to match the exact start of the payload code. In Figure 2.3, the pointer from the overwritten return instruction pointer (RIP) points into the middle of the *NOP slide*. Behind it, the code chosen by the attacker for execution is written onto the stack. This is called a *code injection* exploit.

2.4.2. Data Execution Prevention (DEP)

Data execution prevention (DEP) is a mitigation technique to prevent the execution of code on the stack [19]. With DEP, memory pages can either be executable or writable, but not both. Pages that contain the stack are marked as non-executable. This achieves improved security, because an attacker can not simply write code to the stack using a buffer overflow as described in the previous section and execute it. Modern operating systems use DEP by default [11].

2.4.3. Address Space Layout Randomization (ASLR)

To work around DEP, attackers can reuse existing code in the binary, because it is located on executable (but not writable) memory pages. However, this can be mitigated using Address

Space Layout Randomization (ASLR). This technique loads dynamically-linked libraries into random memory positions to avoid having predictable function addresses [35]. Thus, attackers can not use code that is part of standard libraries as a target of control flow redirection, because they can not know the exact address of the desired function. ASLR does however not apply to statically-linked code, which the Go compiler mostly creates.

2.4.4. Return-Oriented Programming (ROP)

Return-oriented programming (ROP) is a specific technique for reusing code. It suggests jumping to very small so-called gadgets, which are sequences of two or a few more machine instructions that do not modify the stack pointer register and end with a return instruction [44]. With these properties, gadgets can be chained, because after the execution of a fragment the return instruction fetches the next address to jump to from the stack, which an attacker can control when they exploit a buffer overflow vulnerability. Depending on the available ROP gadgets, the attacker can concatenate them and write a program with a limited set of assembler instructions available. Often, there are enough gadgets present in the program code to form a set that together is Turing-complete, thus, arbitrary behavior can be induced from it. The Go standard library is very large and provides many available gadgets. This could make exploiting a buffer overflow, once it exists in a Go program, even easier than in a C program, where an attacker might have to work around ASLR, for example.

2.5. Dependency Management in Go

To break code into logical units that can be maintained separately and allow reusing them, Go offers a dependency management system. The smallest possible unit is a Go package[6]. A package can contain one or more source code files which must all be in the same directory. It has a name which should match the name of the directory containing the source files. There can not be more than one package in the same folder. Identifiers, such as function and variable names, must be unique in the scope of the same package and can be accessed directly. Therefore, developers can split their code into individual code files without restrictions to organize a single package. When accessing code in a different package, the package name must be used as prefix in front of the identifier name. Furthermore, the package must be imported using an *import* statement at the beginning of the file. When importing the package, it is necessary to specify the complete path to the package, the name is not sufficient on its own. The import path is the directory tree leading to the package directory, separated by forward slashes. For example, a valid import path would be *my-application/storage/database*.

[6] `https://golang.org/doc/code.html`

To allow sharing code with other developers and implementing reusable libraries, it is possible to download external packages using the *go get* command, which can fetch packages using the web protocol HTTP, Git, or other means. In this case, the import path usually begins with a DNS host name such as *github.com*, which hosts the package code. This part is called *registry*.

2.5.1. GOPATH and GOROOT

Before the release of Go 1.13, packages that were fetched using *go get* were stored in the *GOPATH* directory[7]. Furthermore, there exists the *GOROOT* directory, which is similar to *GOPATH*, but contains the Go standard library instead of additionally downloaded packages. Environment variables point to these folders, and the Go compiler uses them to get the sources of packages that are not part of the current project.

This architecture provided a simple way of downloading and using external packages. However, a serious drawback is that it does not allow to specify a version of the imported package. Therefore, it is very hard to publish stable libraries, because developers can not depend on an old version of the library, and it is not possible to use different versions of the same package next to each other.

2.5.2. Go Module System

Starting with the release of Go 1.11, the development of the module system has fixed this by introducing modules as a new unit of code organization[8]. The system is stable and used as the default since Go 1.13 in September 2019. Modules contain one or more packages, and have a name and import path similar to packages, which can in fact be the name of a root package in the module. Furthermore, modules are required to be available as a revision control repository, like Git, and are versioned. Downloaded modules are placed in the *module path*, a subdirectory of *GOPATH*.

A project can define dependencies to specific versions of other modules, which resolve to the packages imported in the code. Thus, it is possible to require a specific version of an external package. Dependencies can be transitive, because imported modules can have dependencies to other modules themselves, forming a dependency tree. Figure 2.4 shows an example of the relations between the different units of code organization used in Go. The *github.com/example/example* module is the main application module and its current version is *v2.0.0*. It contains three separate packages *A*, *B*, and *C*, all of which are composed of individual Go source files. The module depends upon two other modules *X* and *Y*, both of which are imported with a specific version. These in turn contain one or more packages with source files, and further depend on more modules. The *Y* module is effectively imported in two different versions, *v1.0.1* and *v5.3.0*, thus both their sources will be compiled and included in the resulting binary.

[7] `https://golang.org/doc/gopath_code.html`
[8] `https://blog.golang.org/using-go-modules`

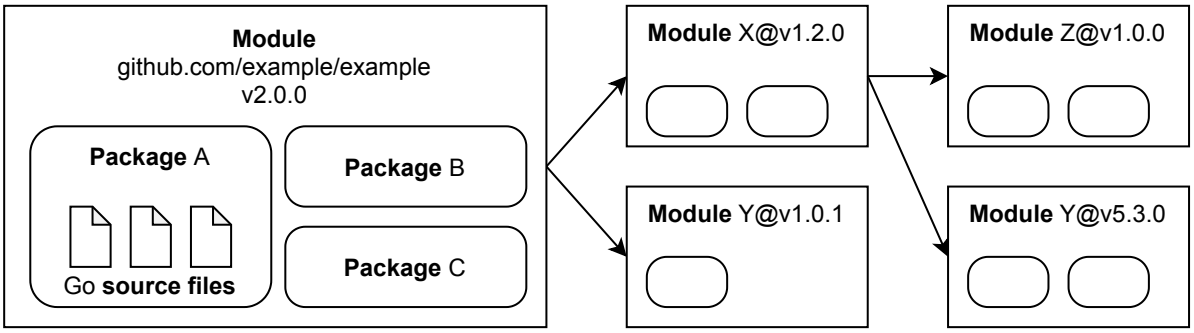


Figure 2.4.: Dependency management model used by Go

Projects define their dependencies including specific versions in a file called *Go.mod*. When the code is built, the Go compiler toolchain will automatically figure out which modules need to be downloaded.

2.6. Static Code Analysis

Static code analysis describes techniques that examine source code for particular properties, such as code metrics or the presence of specific bugs, without running the program. It is different from dynamic code analysis, which executes the code and employs runtime checks. Using static code analysis, it is possible to automatically test if the source code matches certain expectations. Based on the level of abstraction that an analysis tool is looking at, it can be classified into lexical, syntactic, or semantic analysis. Lexical analysis deals with the source code as text, with the most common purpose being coding style enforcement. For syntactic analysis, the code is parsed to obtain an abstract syntax tree (AST), which allows, for example, to do type checking, and reporting code metrics such as the cyclomatic complexity or McCabe metric [55]. Finally, semantic analysis takes care of the control and data flow, which describe what statements in the code can be executed in which order and how data is being transferred between variables. This allows, for example, to detect unused or dead code, or detect whether a variable is used before it is initialized. To do this, a control flow graph (CFG) is used.

2.6.1. Abstract Syntax Tree (AST)

Source code of programming languages is a formal language that follows a grammar, which is its syntax. Syntactically valid programs are words in the formal language and can be represented as the tree of grammar rules used to construct the word. This is called the abstract syntax tree (AST) of the program [9].

Figure 2.5 shows an example AST for a code fragment of just one statement, which is $x := 3 + 5$.

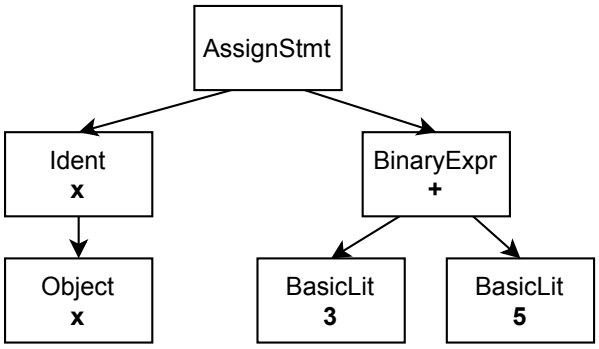


Figure 2.5.: Abstract syntax tree visualization for source code "x := 3 + 5"

A grammar construction of the programming language syntax begins with a start symbol, such as an assignment statement (*AssignStmt*) in this case. It can have children, which are recursively substituted for concrete values. In Figure 2.5, the right hand side of the assignment is an addition, an instance of a binary expression (*BinaryExpr*), with basic integer literals (*BasicLit*) 3 and 5 used for the addition. The AST shows the composition of the source code into logical units of growing size, with the leafs being identifiers, literals, or any other nodes that do not contain further children, such as an empty return statement, for example.

To obtain the AST, it is possible to use the parser bundled with the language's compiler. An advantage of running static analysis steps on the AST is that it is independent of any concrete lexical representation, such as white space. Identifier names can also be easily replaced by different names in the abstract representation, which makes the AST suitable for name refactoring operations or recognizing specific types even if they are aliased in the source code.

2.6.2. Control Flow Graph (CFG)

The control flow graph (CFG) is not purely based on syntax, but needs semantic analysis to be constructed. It is a directed graph showing possible execution paths through code segments. It can contain cycles and is not necessarily connected (neither strongly nor weakly). Nodes in the control flow graph denote logical pieces of the source code in the sense that they form a unit of execution. They can be seen as an atomic step of computation. While CFG nodes are connected to individual nodes in the AST, there are AST nodes that are not directly related to a single CFG node. Edges in the CFG represent all possible paths of execution. An example of a CFG can be seen in Figure 2.6. It shows the simple loop *for* $x < 5$ { $x \mathrel{+}= 1$ }.

There are two logical steps of execution in this code example. After any previous code is executed and control flows to the *for* loop, first the loop condition $x < 5$ is checked. There are two outgoing edges on the node, because the condition can be either true or false. If it is true, control flows to the assignment node representing the $x \mathrel{+}= 1$ statement. That node has an edge back to

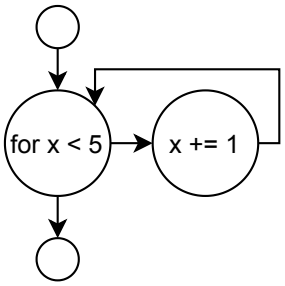


Figure 2.6.: Control flow graph for source code "for x < 5 { x += 1 }"

the loop condition node to check whether the loop should be run again. Otherwise, the code following behind the loop is executed.

Such a CFG is an excellent tool to detect dead code, which would be represented by an isolated or unreachable connectivity component in the graph. Furthermore, it is possible to find control flow anomalies such as *read-before-write* using node annotations, which indicate whether variables are read or assigned. This allows to detect that a variable is used before it is initialized. It is also useful to detect where a variable is assigned for the last time before a particular statement, thus, what value it has at the time of execution of that statement.

2.6.3. Go Linters *go vet* and *gosec*

There are a number of existing static code analysis tools for the Go programming language. The most official is *go vet*[9], which is included as part of the main Go command line tool (Go CLI) and, thus, is installed by default with the Go compiler. It is built using a modular system of individual analysis steps. Examples of these steps include checking if returned values from functions are unused, type checking of the format specifiers used in the *fmt.Printf* function with respect to the concrete parameters, or detecting unreachable code.

Another static analysis tool with a focus on security is *gosec*[10]. Similar to *go vet*, it contains a number of different rules that the source code is checked against, for example, if insecure hash functions are used, template strings contain unescaped data, or whether returned error values from functions are actually checked. Such static analysis tools that detect insecure code or common mistakes are called linters.

[9] `https://golang.org/cmd/vet`
[10] `https://github.com/securego/gosec`

3. Analysis of Security Problems with *Unsafe*

This chapter presents an in-depth security analysis of possible vulnerabilities that can be caused by misuses of the *unsafe* API, and their consequences. Figure 3.1 shows an overview of the contents of this chapter. It is organized into three main areas of danger. First, conversions between types with architecture-dependent memory layout are discussed. Then, incorrect conversions between slices and strings are analyzed with respect to memory management. Finally, buffer overflow bugs and their consequences are shown.

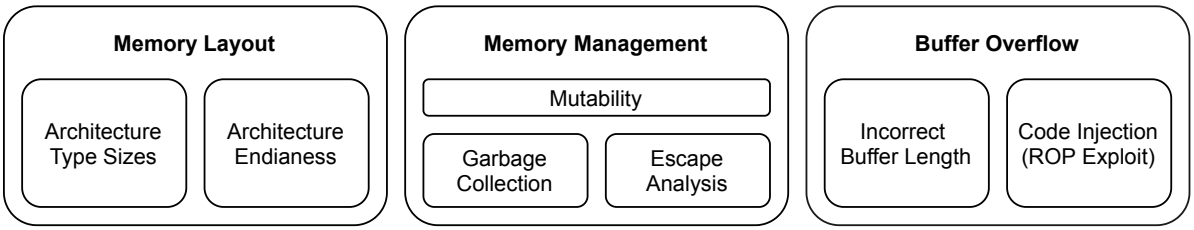


Figure 3.1.: Organization of Chapter 3

3.1. Architecture-Dependent Types

The first area of possible vulnerabilities is about types that have a different size or alignment on various architectures. A feature of Go is portability between different platforms [6]. The same source code can usually be compiled for many architectures without any changes in its behavior. However, this is only true as long as the *unsafe* API is not used. Since *unsafe* code allows direct conversion between arbitrary types, there can be inconsistencies between the sizes and alignments of structure fields on different architectures. An example of this is shown in Listing 3.1. There are two structure types declared, *PinkStruct* (Lines 1–4) and *VioletStruct* (Lines 5–8), both of which have two fields *A* and *B*. While field *B* has the same type (*uint8*) in both types, field *A* is of type *int* (Line 2) and *int64* (Line 6), respectively. An instance of *PinkStruct* is created in Line 11, which is converted to *VioletStruct* in Line 12. This is only a valid operation if *int* and *int64* share the same size and alignment.

Listing 3.1: Incorrect cast between architecture-dependent types

```
1  type PinkStruct struct {
2     A int
3     B uint8
4  }
5  type VioletStruct struct {
6     A int64
7     B uint8
8  }
9
10 func main() {
11    pink := PinkStruct{A: 1, B: 42}
12    violet := *(*VioletStruct)(unsafe.Pointer(&pink))
13 }
```

This is true for 64-bit platforms like *amd64*. If the code is executed on such an architecture, the conversion works fine and the field values in the resulting variable *violet* are as expected. In contrast, if the code is run on an architecture where *int* and *int64* have different sizes or alignments, the values of *violet* are undefined. This is the case, for example, on a 32-bit platform like *i386*. When the structure definitions are not placed directly next to each other but in separate files, packages, or even modules, spotting the difference between the incompatible structs is much harder. On top of that, the bug might never occur in testing but only on specific platforms that get used in production. Possible consequences of a bug like this depend on how the resulting struct value is used in the remainder of the program. If data gets written to it, it changes invalid memory which can lead to a code injection vulnerability, and if it is passed as output it can create an information leak vulnerability.

In addition to incompatible field sizes, there could also be a different byte order on some architecture. In this case, using a direct cast using the *unsafe* API is unaware of possible changes of the byte order, for example, incoming network data and variables stored in the memory could be represented as big and little endians and cause a mismatch. When such data is stored as the length of a slice, the number could be misinterpreted and, thus, a slice with a wrong length could be created. This in term can cause a buffer overflow vulnerability that can lead to code injection.

Insight 1

Different type sizes and byte order on various architectures, used with direct conversions through *unsafe.Pointer*, can cause invalid memory access such as buffer overflows.

3.2. Incorrect Slice and String Casts

The second main area of danger with *unsafe* usages is focused around the conversion of slices of different types, and strings. Listing 3.2 shows a common *unsafe* usage pattern that takes a string argument and converts it into a slice of bytes without allocating additional memory. This allows to access the raw bytes contained in the string in an efficient manner.

Listing 3.2: Conversion from string to []byte 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 }
```

First, the string is converted into the *reflect.StringHeader* representation in Line 2. Then, a slice header is created as a composite literal and the reference to the underlying data array and length information are copied from the string header (Lines 3–7). The capacity is set to the same value as the length, as it would be in a slice that uses the full capacity. Finally, the slice header is converted into an actual *[]byte* slice, which is returned (Line 8).

There are three non-obvious but potentially dangerous problems that come with this type of slice creation, which are described in the following subsections.

3.2.1. Implicit Immutability

The first problem is that the resulting slice is implicitly immutable, or read-only. Regularly, slices in Go are mutable and strings are not. As described in Section 2.2, this is because the underlying string data array is usually located on a read-only memory page. If a developer writes code that changes parts of a string in-place, the compiler will not accept that code and, thus, prevent the possible segmentation fault.

The slice returned by the *StringToBytes* function in Listing 3.2 is mutable. Code that changes the resulting slice will be accepted by the compiler, because it is not able to infer its connection to a former string. However, since the slice uses the same underlying data array as the original string used to do, the data is still potentially located within read-only memory, and modifying it will crash the program. It is only implicitly read-only, but the runtime is not aware of it.

While this first problem of the conversion pattern does not cause a direct security vulnerability, it underlines that there are many consequences of using the *unsafe* API that are not obvious.

Although it is possible in theory to manually ensure that no usages of the *StringToBytes* function modify the slice it returns, in practice it is very hard to do so, especially if new developers join the project after some time and start using the function without having all possible consequences in mind.

Insight 2

Using in-place conversions using the *unsafe* API can create fragile code with implicit rules that all developers working on the code need to follow to maintain correctness.

3.2.2. GC Race Use-After-Free

The second problem with the conversion shown in Listing 3.2 is a *use-after-free* bug caused by a race condition involving the garbage collector (GC) used by the Go runtime. As discussed in Section 2.3.2, the GC treats all pointer types and the special *Data* field of actual slices and strings as reference types. Importantly though, it does not treat normal *uintptr* values and *Data* fields of artificially constructed slice and string headers as references although they contain addresses. This means that converting a *uintptr* value to an *unsafe.Pointer* value is an invalid and dangerous operation. If the *GC* runs while the pointer has not been constructed, it does not mark the value at the address stored in the *uintptr* value as live and thus frees the memory. Creating an *unsafe.Pointer* from this address therefore creates a potentially dangling pointer, because it points to memory that might have been freed. This is a *use-after-free* bug.

Listing 3.2 contains this bug. The *Data* field of the *reflect.SliceHeader* structure created in Line 3 is of type *uintptr*. The header value is not derived by cast from a real slice. Therefore, it does not benefit from the special case built into the Go runtime, which treats *uintptr* references that are part of slices and strings as references. Specifically, the problem is that the slice header has been constructed as a composite literal. Therefore, after Line 7 the *bytesHeader* variable becomes a plain *uintptr* value, and the original string *s* is not used any longer. However, at this point there is no real slice created yet as this happens not earlier than in Line 8. The underlying data array of the string *s* is therefore not a live value as far as the GC is concerned. If the GC runs at this time, the underlying data array used in the slice is already collected when the resulting slice is created. This is possible because the GC runs concurrently to the application logic and can trigger at any time. If the Go program has several threads that all contribute to the heap usage, this is even more severe.

Figure 3.2 illustrates the vulnerability. The green boxes show the original string value and its header (Line 2 in Listing 3.2), which is represented at position 1 in the memory schema. The reference to the underlying data of the string and resulting slice (Line 4) is shown in red at memory position 2. Then, the slice header is created (Lines 3–7), which is colored blue and located at position 3 in the memory. References indicated by arrows are strong if the arrow is

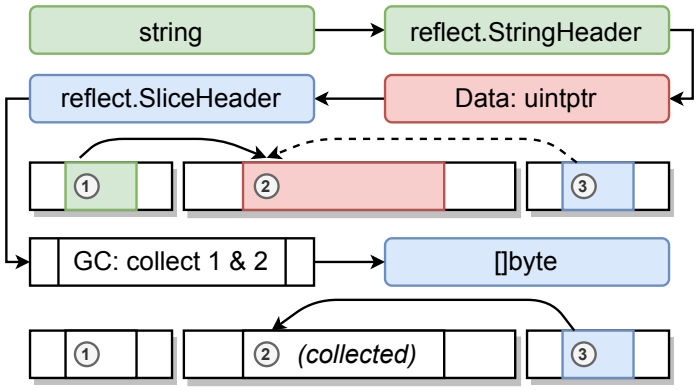


Figure 3.2.: GC race and escape analysis flaw

continuous, like the reference the string header has, and weak if the arrow is dashed, like the new slice header has. Then, after the *GC* runs, there is only the resulting bytes slice left, which is also shown in blue. It now has a strong reference to the underlying data array (at position 2), but since the GC has already run while there was only a weak reference to it, the data has been collected. The resulting bytes slice (Line 8) is now a dangling pointer into the former string data in memory, and the *use-after-free* bug is complete.

If the bytes slice is modified, arbitrary memory is changed. This can allow an attacker to tamper with application data, or possibly change stored return addresses on the stack, therefore achieving a code injection vulnerability.

3.2.3. Escape Analysis Use-After-Free

The third problem that comes with the incorrect construction of a new slice shown in Listing 3.2 is that the Go compiler's escape analysis (EA) algorithm fails. For a similar reason to the GC, it misses the connection between the string parameter and the slice return value. Listing 3.3 shows a proof of concept of this problem.

In the function *GetBytes*, a string variable *s* is created by reading a string through a buffered reader (Lines 9–10). It is necessary to use the reader instead of a string literal, because with a literal the effects of the vulnerability would not be visible. This is however no restriction to the generality of this proof of concept. Creating the string from a reader is similar to accepting dynamic user input. Next, the string is converted to a byte slice using the insecure *StringToBytes* function (Listing 3.2) in Line 11. Printing the resulting slice in Line 14 yields the expected output equal to the original string, which is *abcdefgh*. Finally, the slice is returned to the caller function, which in this case is the *main* function in Line 2. The slice is printed again in the *main* function in Line 5, and while the expected output would be identical, this time invalid, random data is printed.

Listing 3.3: Escape analysis flaw proof of concept

```
1  func main() {
2     bytesResult := GetBytes()
3     // expected stdout is "abcdefgh"
4     // actual output is random invalid data
5     fmt.Printf("main: %s\n", bytesResult)
6  }
7
8  func GetBytes() []byte {
9     reader := bufio.NewReader(strings.NewReader("abcdefgh"))
10    s, _ := reader.ReadString('\n')
11    out := StringToBytes(s)
12    // expected stdout is "abcdefgh"
13    // actual output is "abcdefgh"
14    fmt.Printf("GetBytes: %s\n", out)
15    return out
16 }
```

The reason for this is that when the Go EA algorithm determines where to allocate the string *s* in the *GetBytes* function, it detects that *s* is passed to the *StringToBytes* function and therefore transitively analyzes that function. Within the function, *s* is used in a cast to the *strHeader* variable in Line 2 of Listing 3.2. After that, *strHeader* is used to construct the *bytesHeader* variable, but at this point there are no further references to the underlying data array of *s*. The reason for this is that both *s* and *strHeader* are not used further in the function. Similarly to the GC mark algorithm, the *Data uintptr* field would only be treated as a reference with respect to escape analysis if the slice header had been created by cast from an actual slice. Therefore, the EA determines that *s* does not escape in *StringToBytes*, and since it is not used any longer in *GetBytes*, it does not escape at all and is placed on the stack of the *GetBytes* function. The *EA* algorithm fails to detect that there is a reference to the string *s* in the *out* variable, which obviously escapes as it is returned.

Printing the *out* slice in Line 14 works, because the stack of the *GetBytes* function still exists at this point. However, when the function returns, the stack frame is removed, and *bytesResult* in Line 2 becomes a dangling pointer into the former stack of *GetBytes*. This is a *use-after-free* bug, because that memory can and probably will be reused at that point. The print statement in Line 5 outputs invalid data for this reason. If the *main* function would modify the contents of the *bytesResult* slice, it would change arbitrary memory, with possible consequences such as information leaks or code injection.

Insight 3

Creating slice or string headers from scratch is an invalid and dangerous operation, because the Go GC and EA do not recognize the reference to the underlying array in memory. This causes *use-after-free* bugs for multiple reasons, and can result in access to arbitrary memory and code injection for an attacker.

3.3. Buffer Overflow Vulnerabilities

The third main area of possible vulnerabilities in the context of the *unsafe* API are buffer overflows. In the following subsections, different ways of creating buffer overflows are discussed, and a concrete example of using a vulnerability with return-oriented programming (ROP) is given.

3.3.1. Incompatible Types

Converting slices to strings and vice versa can be done simply by casting them, however this will cause the Go runtime to allocate a new slice or string for the resulting value. To improve efficiency by reusing the underlying data and only reinterpret it as a new type, it is possible to use the *unsafe* API for an in-place cast. Since the fields in the header structures shown in Listing 2.3 in Section 2.2 are different in the presence of the *Cap* field, this is however only valid for conversions from slices to strings. In that case, the resulting string header, which shares the same location in memory with the original slice header, will reuse the reference to the underlying data array as well as the length, and since the string header structure does not contain any more fields the capacity information will be unused although it remains in adjacent memory.

In contrast, when converting a string to a slice directly, the source header structure is too short. The resulting slice header will use correct values for its data reference and length, but the capacity field will contain whatever is located in memory directly after the original string header. This is shown in Listing 3.4, which is taken from the *k8s.io/apiserver* package that is part of the *Kubernetes* project[1].

Listing 3.4: Incorrect direct cast from string to slice (code from k8s.io/apiserver)

```
1 // toBytes performs unholy acts to avoid allocations (sic)
2 func toBytes(s string) []byte {
3     return *(*[]byte)(unsafe.Pointer(&s))
4 }
```

[1] https://github.com/kubernetes/apiserver

The resulting slice (Line 3) is dangerous to use, because it contains random data in its capacity field. Operations that read from a slice, like using it in a loop to iterate over its contents, mostly only use the data reference and length. However, any action on the slice that uses the capacity, like taking subslices from it or appending data, is undefined and can therefore become a security vulnerability. A slice with a capacity that is greater than the actual allocated memory can be used to read and write adjacent memory by increasing its length until it reaches outside its valid bounds. Thus, an attacker can control invalid memory, leading to possible code injection or information leak.

Insight 4

Operations that seem to be valid in two directions, and in fact look symmetric both ways, can be incorrect in one of the directions. Using the *unsafe* API, this can create an exploitable buffer overflow vulnerability.

3.3.2. Incorrect Length Information

Buffer overflows can also be caused by creating a new slice that has an incorrect length with respect to its underlying data array. This can happen quickly if the length information is calculated dynamically, especially if user-supplied input data is used to deduce the resulting value. An example of such a bug is shown in Listing 3.5. It is taken from the *hanwen/go-fuse* library[2], a Go implementation of the server specification used for the userspace file system (FUSE) available for Linux[3]. A patch for this bug has already been submitted to the authors.

FUSE uses a client / server architecture where file system implementations present a server that handles requests from a specific kernel module. The listing shows a part of the *doBatchForget* function, which is used to clear files, identified by their inode number, from a cache that the file system might have. In the design of *go-fuse*, incoming data from the */dev/fuse* node is passed to handler functions as byte slices (*req.inData* and *req.arg* in Lines 2 and 5) containing the raw, serialized request. The handler function then casts the data into a suitable structure to access individual fields. In this case, the request is converted into an instance of the *_BatchForgetIn* type (Line 2), which provides access to the number of inodes to forget in the batch request through the *in.Count* field (Line 3). The individual forget messages in serialized and concatenated form are available through the *req.arg* array, so to access them a new slice is created by defining its slice header from scratch (Lines 11–15). The created slice uses the incoming request data containing the forget messages (*req.arg*, Line 12) as its underlying data array, and the number of messages (*in.Count*) both as length and capacity (Lines 13 and 14).

Since the *in.Count* field is supplied as part of the request sent to the library, it can be controlled

[2] `https://github.com/hanwen/go-fuse`
[3] `https://www.kernel.org/doc/html/latest/filesystems/fuse.html`

Listing 3.5: Incorrect slice length bug in the hanwen/go-fuse library

```
1  func doBatchForget(server *Server, req *request) {
2     in := (*_BatchForgetIn)(req.inData)
3     wantBytes := uintptr(in.Count) * unsafe.Sizeof(_ForgetOne{})
4
5     if uintptr(len(req.arg)) < wantBytes {
6        // We have no return value to complain, so log an error.
7        log.Printf("Too few bytes for batch forget.",
8                len(req.arg), wantBytes, in.Count)
9     }
10
11    h := &reflect.SliceHeader{
12       Data: uintptr(unsafe.Pointer(&req.arg[0])),
13       Len: int(in.Count),
14       Cap: int(in.Count),
15    }
16    forgets := *(*[]_ForgetOne)(unsafe.Pointer(h))
17    // ...
18 }
```

by an attacker if they manage to conduct a man-in-the-middle attack and change the request to be a malicious one. If the length is greater than the available number of data bytes, the resulting *forgets* slice (Line 16) will access additional data after the end of the actual *batch forgets* request. Depending of the concrete operation to prune inodes from a local cache, this could have severe consequences with respect to security. To prevent this from happening, the authors intended to check the length of the available data in Line 5, and if there are too few bytes for the alleged number of individual requests, the method should not proceed. However, as the listing shows there is no actual code to stop further execution of the method, instead there is only a log message noting that there was a malformed request with too few bytes available (Lines 7–8).

A proof-of-concept implementation[4] of a potential exploit showed that the method actually accesses invalid memory data located after the request data. Depending on the code that calls this function, an attacker can use this to read or write this memory, again possibly enabling arbitrary code injection or information leak exploits. In this case, a simple *return* statement after Line 8 would have prevented this bug.

Insight 5

In Go code that uses the *unsafe* API, developers who fail to implement proper bounds checking can quickly cause very serious security vulnerabilities. Code then has to be written and audited as carefully as traditional, non-memory-safe C code.

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

3.3.3. Code Injection using ROP

This section shows practical evidence that buffer overflows as described in the previous sections can actually cause an attacker to achieve code injection. This is done by outlining a proof-of-concept exploit using return-oriented programming (ROP), the source code of which is available on *GitHub*[5].

Listing 3.6 shows a short example function that contains a buffer overflow. In Line 4, a byte slice with a length of 512 bytes is initialized. Then, its slice header is obtained in Line 5 and used to change the underlying data array of the slice (Line 6). After the switch, the slice is backed up by a much shorter byte buffer of eight bytes (initialized in Line 2).

Listing 3.6: Buffer overflow leading to code flow redirection

```
1 func main() {
2     harmlessData := [8]byte{'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A'}
3
4     confusedSlice := make([]byte, 512)
5     sliceHeader := (*reflect.SliceHeader)(unsafe.Pointer(&confusedSlice))
6     sliceHeader.Data = uintptr(unsafe.Pointer(&(harmlessData[0])))
7
8     _, _ = bufio.NewReader(os.Stdin).Read(confusedSlice)
9 }
```

Then, a reader is used to read input data into this malicious slice in Line 8. Since it has a length of 512 bytes, the function reads up to this many bytes, however because the actual data array used by the slice is much shorter, this will overflow into the memory behind the data array. In this case, the *harmlessData* array is created on the stack of the function, which means it is vulnerable to stack-based buffer overflow exploits as described in Section 2.4.1. An attacker can use this vulnerability to redirect the code flow by carefully crafting the input data such that the saved return instruction pointer on the stack is overwritten with the address of a function of the attacker's choice.

As described in Chapter 2, DEP prevents directly putting exploit code on the stack, but since the Go compiler uses static linking, ASLR is not much of a concern. Using ROP, it is possible to use gadgets with machine instructions that are part of the Go standard library, which is linked into the binary of all Go applications. A strategy to exploit a buffer overflow to achieve code injection is to use the *mprotect* and *read* system calls. A system call, or syscall, is a function provided by the operating system, which can be executed using the *syscall* machine instruction. The processor registers control which syscall gets executed and which parameters it receives. Therefore, an attacker needs ROP gadgets to control the contents of the registers, as well as to trigger the syscall. There are several alternative gadgets for this available in the Go standard library.

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

Using the *mprotect* call, the attacker can set a memory area of their choice to have both writable and executable flags, for example, the heap area. Then, using the *read* call, they can store data in that memory section, which is usually an exploit payload spawning a shell to receive further arbitrary commands. The payload is provided as part of the input string created by the attacker. Finally, by using the address of the code area just like an address of another ROP gadget the attacker can make the processor jump to the payload code. Because in contrast to the stack the code is located on an executable page, the CPU executes it and the code injection attack succeeds. This clearly demonstrates the severeness of buffer overflows introduced through misuses of the *unsafe* API in Go programs with concrete evidence on how to create a successful exploit.

Insight 6

Once a Go program contains a buffer overflow vulnerability, established and well-researched exploit techniques like ROP can be used just as in traditional languages like C to achieve code injection.

3.4. Summary

This chapter presented multiple different ways of misusing the *unsafe* API in Go applications. Almost all of them can be used, in one way or another, to effectively inject and execute arbitrary code, or read and modify protected data within the process memory.

Many of the dangers are correlated with slices and strings, and especially their internal structure that is accessible through their header structures. This is not surprising since slices are a close equivalent to buffers in other languages, which means they are often concerned with user-supplied data, and inherently carry the danger of missing proper bounds checking.

More details and proof-of-concept exploit code for the vulnerabilities presented in this chapter are available in a dedicated repository on *GitHub*[6].

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

4. *go-geiger*: Identification of *Unsafe* Usage

This chapter presents *go-geiger*, a tool to find usages of the *unsafe* API in Go packages and their dependencies. Figure 4.1 shows an outline of the contents of this chapter. Using open-source projects from *GitHub* and *go-geiger*, an empirical study on *unsafe* usage is done, which is used additionally to quantitatively evaluate *go-geiger*. Furthermore, a novel data set of labeled samples of *unsafe* code is described and used as a qualitative analysis on *unsafe* usage in Go.

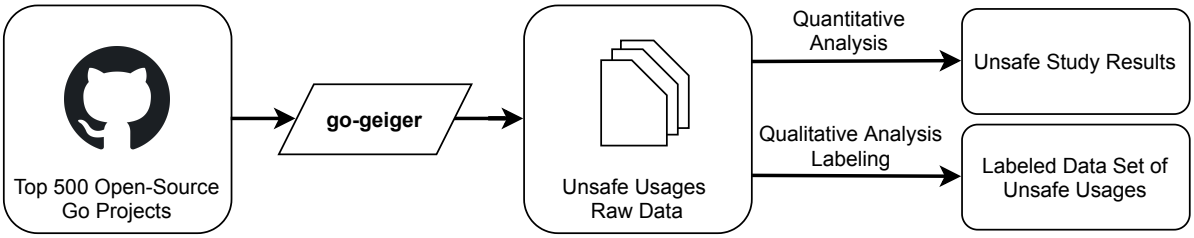


Figure 4.1.: Organization of Chapter 4

4.1. Design

The novel tool *go-geiger* is designed to identify usages of the *unsafe* API in Go source code. It is available on *GitHub*[1]. The tool includes the dependencies of Go packages in its analysis, which gives a more complete picture of possible *unsafe* usages than only looking at an individual package. It is inspired by *cargo geiger*[2], a similar tool for detecting the use of *unsafe* code blocks in Rust programs.

Figure 4.2 shows the architecture of *go-geiger*. First, it determines the scope of code that should be analyzed for *unsafe* usage. To achieve this, the dependency tree of the packages given to *go-geiger* is built. Then, the sources of all the packages in this tree are parsed, and the resulting abstract syntax trees (AST) are inspected. Within the AST, usages of the *unsafe* API are identified. Each usage is assigned a tuple of labels consisting of *match type* and *context*

[1] `https://github.com/jlauinger/go-geiger`
[2] `https://github.com/rust-secure-code/cargo-geiger`

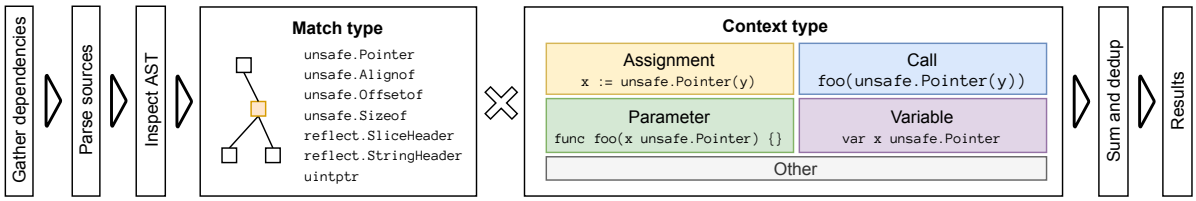


Figure 4.2.: Architecture of the go-geiger tool to detect unsafe usages

```
 WITH DEPENDENCIES   LOCAL PACKAGE   VARIABLE   PARAMETER   ASSIGNMENT   CALL   OTHER   PACKAGE PATH
-------------------+---------------+----------+-----------+------------+------+-------+-----------------------------------------------------------------
        405                0           0           0           0          0      0     github.com/jlauinger/go-geiger
        405                0           0           0           0          0      0     └─github.com/jlauinger/go-geiger/cmd
        405                0           0           0           0          0      0       ├─github.com/jlauinger/go-geiger/counter
        398                0           0           0           0          0      0       │ ├─github.com/fatih/color
        398                0           0           0           0          0      0       │ │ ├─github.com/mattn/go-colorable
        398                0           0           0           0          0      0       │ │ │ └─github.com/mattn/go-isatty
        398              398          38          25         327          8      0       │ │ │   └─golang.org/x/sys/unix
        398                0           0           0           0          0      0       │ │ └─github.com/mattn/go-isatty...
          0                0           0           0           0          0      0       │ ├─github.com/olekukonko/tablewriter
          0                0           0           0           0          0      0       │ │ └─github.com/mattn/go-runewidth

Package github.com/jlauinger/go-geiger including imports effectively makes up 24 packages
  3 of those contain unsafe.Pointer usages
  12 of those further import packages that contain unsafe.Pointer usages
  9 of those do not contain any unsafe.Pointer usages

Packages in green have no unsafe.Pointer usages
Packages in red contain unsafe.Pointer usages
Packages in white import packages with unsafe.Pointer usages
```

Figure 4.3.: Usage example screenshot of go-geiger

type. The match type represents the part of the *unsafe* API that is used. It can be one of the four *unsafe* package members *Pointer*, *Alignof*, *Offsetof*, and *Sizeof*, the *reflect* package fields *SliceHeader* and *StringHeader*, or the *uintptr* keyword. The context type indicates the functional part of code that the usage is found in. It can be either an *assignment*, a *call* to a function, a function *parameter* definition, or a *variable* definition. Context identification is done based on the abstract syntax tree (AST), which might lead to edge cases in some programs that are not handled by *go-geiger*. For example, this could happen if future releases of Go introduce new AST node types. To mitigate this possibility, the context is set to *other* in *go-geiger* if all four other options can not be applied. The context allows to filter the search for *unsafe* usages in particular functional code positions. For example, it is possible to find only *unsafe* usages that are used within parameters of function calls. After the *unsafe* usages are identified, they are counted. For this, it is necessary to take care of package deduplication. If a particular package exists in the dependency tree multiple times and is reachable on different paths, it must still be counted only once when calculating the sum of *unsafe* usages in a package's dependencies. This is because the package does not get any less safe by including the same code multiple times. The code is already part of the resulting program. Finally, the analysis results are shown to the user.

Figure 4.3 shows a screenshot of *go-geiger*. It presents the analysis results for the *go-geiger* source code itself. However, to decrease the space needed to display the figure, parts of the table with

unsafe counts have been excluded. The output is composed of three sections. The first is a table showing the *unsafe* usage counts for each package in the dependency tree for the packages given for analysis (Lines 1–12). The first column indicates the total number of usages in the package including all its dependencies. The second column shows the number for the package itself, without the dependencies. The heading for this column is *local package*. Next, there are five columns with individual usage counts for the possible context types described above. These columns add up to the *local package* count. Finally, the import path for the package is given to identify it. Lines that are printed in green (Lines 11 and 12 in Figure 4.3) represent packages with no local *unsafe* usages. Red lines (Line 9) are packages that directly contain *unsafe* code, and white lines (e.g. Line 3) indicate that the package does not contain *unsafe* usages itself, but introduces them through its dependencies. After the table, a summary of the number of packages belonging into these three categories is shown (Lines 13–16). The output concludes with a legend for the colors (Lines 18–20).

4.2. Implementation

The identification of the dependency trees for the packages to analyze, as well as the parsing of the source code, is done using the standard Go compiler toolchain. It is accessible using the *packages* API[3]. Similarly, the inspection of the AST is done using the API available through the *ast* package in Go[4].

To find *unsafe* usages corresponding to the different match types shown in Figure 4.2, the AST is filtered for selector expressions (*SelectorExpr* nodes) and identifiers (*Ident* nodes). The first indicate possible usages of *unsafe.Pointer*, *Alignof*, *Offsetof*, *Sizeof*, *reflect.SliceHeader*, or *StringHeader*. The second is used to find instances of *uintptr*. In both cases, the concrete identifier names in the AST nodes are checked to distinguish *unsafe* usages from arbitrary other field accesses. The context type is determined by going up in the AST starting at the expression node corresponding to a given *unsafe* usage. For example, an *unsafe* usage is considered part of the *assignment* context type if it is a descendant of either an assignment statement (*AssignStmt* node), composite literal (*CompositeLit*), or return statement (*ReturnStmt*). The *call*, *parameter*, and *variable* classes correspond to call expression (*CallExpr*), function declaration (*FuncDecl*), and general declaration (*GenDecl*) nodes, respectively. To achieve an effective assignment of the context type, the order in which the different types are checked is important. This is because an *unsafe* usage could be included in several context types in an example like *x := f(unsafe.Pointer(y))*. The AST for this statement consists of an assignment statement, where the right hand side is a call expression. This call expression contains the *unsafe* usage. Therefore, by ascending in the AST and depending on the order of checking, both context types *call* and

[3] `https://pkg.go.dev/golang.org/x/tools/go/packages`
[4] `https://golang.org/pkg/go/ast/`

assignment could be applied. Which order should be used is a design decision, and for *go-geiger* it is *assignment*, *call*, *parameter*, and finally *variable*. Thus, the example above would be classified as *assignment*.

The *unsafe* usage counts are first collected individually for every package. A cache is used to avoid analyzing the same package multiple times if it is present multiple times in the dependency tree. This cache is aware of possibly different versions of the same package and stores them separately. When the usage count including dependencies is calculated, *go-geiger* starts with the root packages that were requested for analysis by the user, and recursively calculates the respective counts. Again, the cache is used to avoid summing up multiple times. This approach is a depth-first traversal of the dependency tree.

4.3. Quantitative Evaluation

To evaluate *go-geiger* on real-world code, it is used to gather empirical data about the usage of *unsafe* in popular open-source Go projects. This section presents a study which was designed to answer the following research questions:

RQ1 How prevalent is *unsafe* in Go projects?

RQ2 How deep are code packages using *unsafe* buried in the dependency tree?

RQ3 Which *unsafe* keywords are used most?

RQ4 Does *unsafe* usage correlate to project metrics such as age or popularity?

RQ5 How does the use of *unsafe* change over the lifetime of the code?

RQ6 Does *go-geiger* provide additional insights into *unsafe* usage compared to other linters?

RQ7 Which *unsafe* operations are used in practice, and for what purpose?

The following subsections first discuss how the data for this study was gathered, and then answer research questions RQ1 to RQ6. Section 4.4.2 presents the answer to RQ7. It is answered later, because it is based on a novel, manually labeled data set of *unsafe* usage purposes, which is introduced in Section 4.4.1.

4.3.1. Data Set

To build a data set of *unsafe* usages, first the top 500 most-starred open-source Go projects available on *GitHub* were downloaded. This initial download was done on May 27, 2020. These 500 projects with their respective revisions are listed in the appendix (Table A.1). Since *go-geiger* is specifically built to analyze the project dependencies, all projects that do not support the Go modules system were removed from the set. This is necessary to ensure that all dependencies

can be automatically resolved and the respective sources are available for analysis. There were 150 projects that had no support for modules. They are indicated in Table A.1 by the *nm* label in the last column. Furthermore, seven projects could not be compiled and thus also had to be excluded from the data set. The *bf* label is used in the appendix table to mark those projects. Since *go-geiger* analyzes the AST, it can not work on packages that can not be parsed. This results in a set of 343 Go projects. These have between 3,075 and 72,988 stars, with an average of 7,860.

As a next step, the dependency trees for all projects selected for analysis were built, resulting in 62,025 unique packages. These consist of 186 packages that are part of the Go standard library, and 61,839 which are not. The average size of the packages is 1,402 lines of code (LOC), with a standard deviation of 5,612, in 4.4 Go files on average (standard deviation 11.3). These packages were analyzed using *go-geiger*, as well as the existing static analysis tools *go vet* and *gosec* to allow a comparison between the findings of these tools. The resulting findings were stored in machine-readable CSV files. The data set contains 199,496 unique *unsafe* usages. All data files as well as the data acquisition scripts used to download the projects and run the analysis tools are available in a data repository on *GitHub*[5]. Additionally, the data repository[6] and the raw projects source code[7] are published on *Zenodo*.

4.3.2. *Unsafe* Usages in Projects and Dependencies (RQ1, RQ2, RQ3)

To attribute *unsafe* usages to either a project or one of its dependencies, the root module of the project is used. Since the data set was constructed such that all projects support the Go module system as described in the previous section, there is a top-level *go.mod* file present for each of them. The module specified in that file is stored as the project root. Packages that are part of this module are (first-party) project code. Other packages that are present in the dependency tree, but not in the root module, are (third-party) dependencies.

By looking at *unsafe* findings that are part of first-party packages, it is possible to determine how many projects directly use *unsafe* in their code. The data shows that this is the case for 131 (38.19%) of the 343 projects. However, this does not take the dependencies into account. With respect to these, 312 (90.96%) of the projects transitively import *unsafe* usages. To calculate this percentage, the complete dependency trees of the projects are integrated into the analysis. This results in 62,025 unique packages, 3,388 (5.5%) of which contain at least one *unsafe* usage. This answers research question RQ1 about the prevalence of *unsafe* in Go projects.

The fraction of projects importing *unsafe* in the previous paragraph does not include the Go standard library. All analyzed projects import this library and it contains *unsafe* usages, which means that with it 100% of the projects would transitively use *unsafe*. Because the standard

[5] `https://github.com/stg-tud/unsafe_go_study_results`
[6] `https://zenodo.org/record/4130780`
[7] `https://zenodo.org/record/4001728`

library is developed by the Go core team, it can be assumed that it is well audited and reasonably safe to use. Since there is no way to exclude it from a project anyways, the analysis presented in this study is more meaningful when the standard library is not causing a project to be counted as using *unsafe*.

Answer to RQ1

About 38% of projects contain *unsafe* usages in their first-party code. Approximately 91% of projects transitively import at least one third-party dependency package with *unsafe* usages.

To answer research question RQ2 about how deep in the dependency tree packages using *unsafe* are usually located, the import depth of all packages used by a project is determined. Import depth denotes the minimum depth of a package in the dependency tree of a particular project, which is the shortest path from the project root module to the package. Thus, all packages included in the root module have an import depth of zero. Packages which are imported by those have a depth of one, and so on. The depth is calculated using breadth-first search on the dependency tree, which saves time when packages are imported multiple times, because the analysis is focused on the minimum. Figure 4.4 presents a heat map plot of the number of packages containing *unsafe* usages by their import depth. The y-axis denotes the depth, the color intensity shows the number of packages using *unsafe* at a given depth, and the x-axis represents the 343 analyzed projects. On the left hand side, next to the heat map, the horizontal bar chart visualizes the total number of packages using *unsafe* at each import depth summed up over all projects.

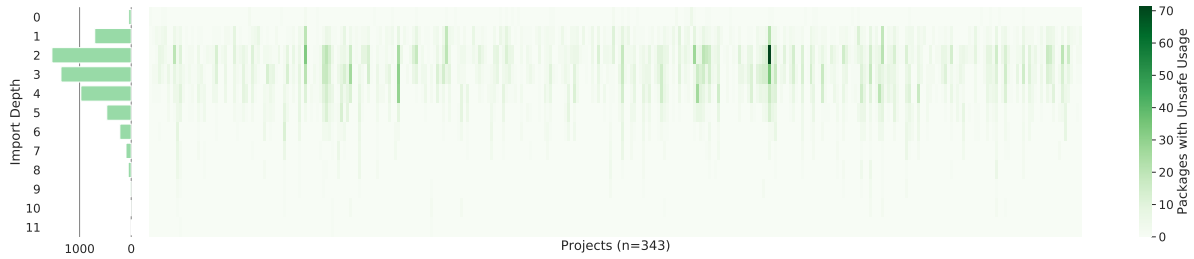


Figure 4.4.: Import depth of packages with unsafe in dependencies of analyzed projects

Figure 4.4 shows that most packages containing *unsafe* usages are imported fairly early, however not directly at the first level. The average depth is 3.08 with a standard deviation of 1.62. The general import depth of all packages, whether they contain *unsafe* or not, is only slightly lower at 3.04. While the depth numbers are fairly low, the total count of imported packages at each level of import depth increases exponentially. Thus, while being possible, it is hard for developers to manually audit dependency packages for *unsafe* usages. The novel *go-geiger* tool helps by quickly

identifying the packages containing *unsafe*, therefore, it is possible to conduct a focused review of the *unsafe* code. Only the first level of dependencies contains the packages that the project developers added themselves, thus, they are obvious to them. In the data set, 496 (14.6%) of the 3,388 packages containing *unsafe* are imported at level one of the dependency tree, and 51 (1.5%) are not imported (level zero). This further shows that the majority of *unsafe* code is introduced further down in the dependency tree.

Answer to RQ2

Most imported packages with at least one *unsafe* usage are located around an average depth of 3 in the dependency tree.

Research question RQ3 is about which *unsafe* tokens are used the most. As described in Section 4.1, *go-geiger* identifies usages of the four members of the *unsafe* package, the *reflect.SliceHeader* and *reflect.StringHeader* types, and *uintptr*. Figure 4.5 shows the distribution of these *unsafe* types in the data set of Go projects.

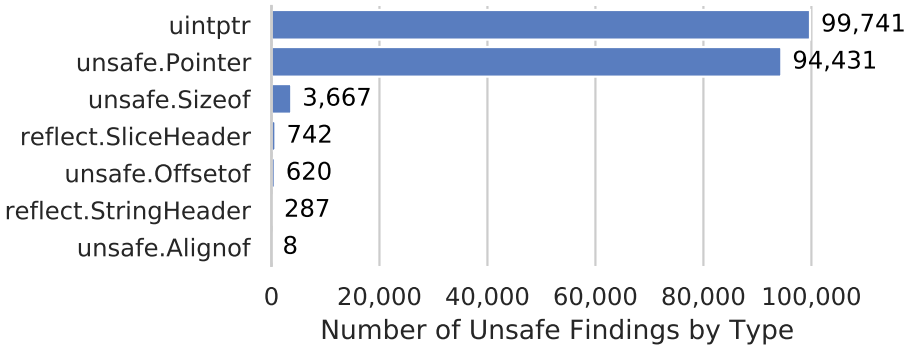


Figure 4.5.: Number of unsafe findings of different types

The data shows that *uintptr* is the most common *unsafe* token, with 99,741 findings. Next, *unsafe.Pointer* has a similarly high prevalence of 94,431 samples. These two lead the usage counts by far, with the next being *unsafe.Sizeof* at only 3,667 usages, and all other token types found less than 1,000 times. With a mere 8 usages found, *unsafe.Alignof* is the most rare.

Answer to RQ3

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

4.3.3. Influence of Age and Popularity (RQ4)

This subsection answers research question RQ4 about whether there is a correlation between *unsafe* usage and the common project metrics age and popularity. Popularity is measured by the number of stars and number of forks that a project has on *GitHub*. Figure 4.6 presents a scatter plot showing the number of *unsafe* usages on the x-axis and the project metrics stars, forks, and age on the y-axis. Usages that are part of the Go standard library are not counted in this graph. Each data point represents one project. Blue circles indicate a project's number of stars, orange diamonds show the number of forks, and green crosses denote the age.

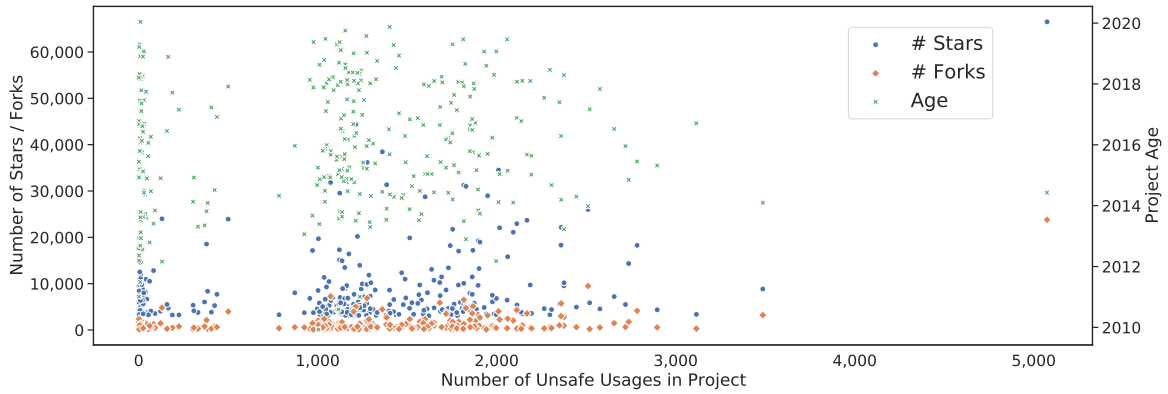


Figure 4.6.: Correlation between unsafe usage and project metrics age and popularity

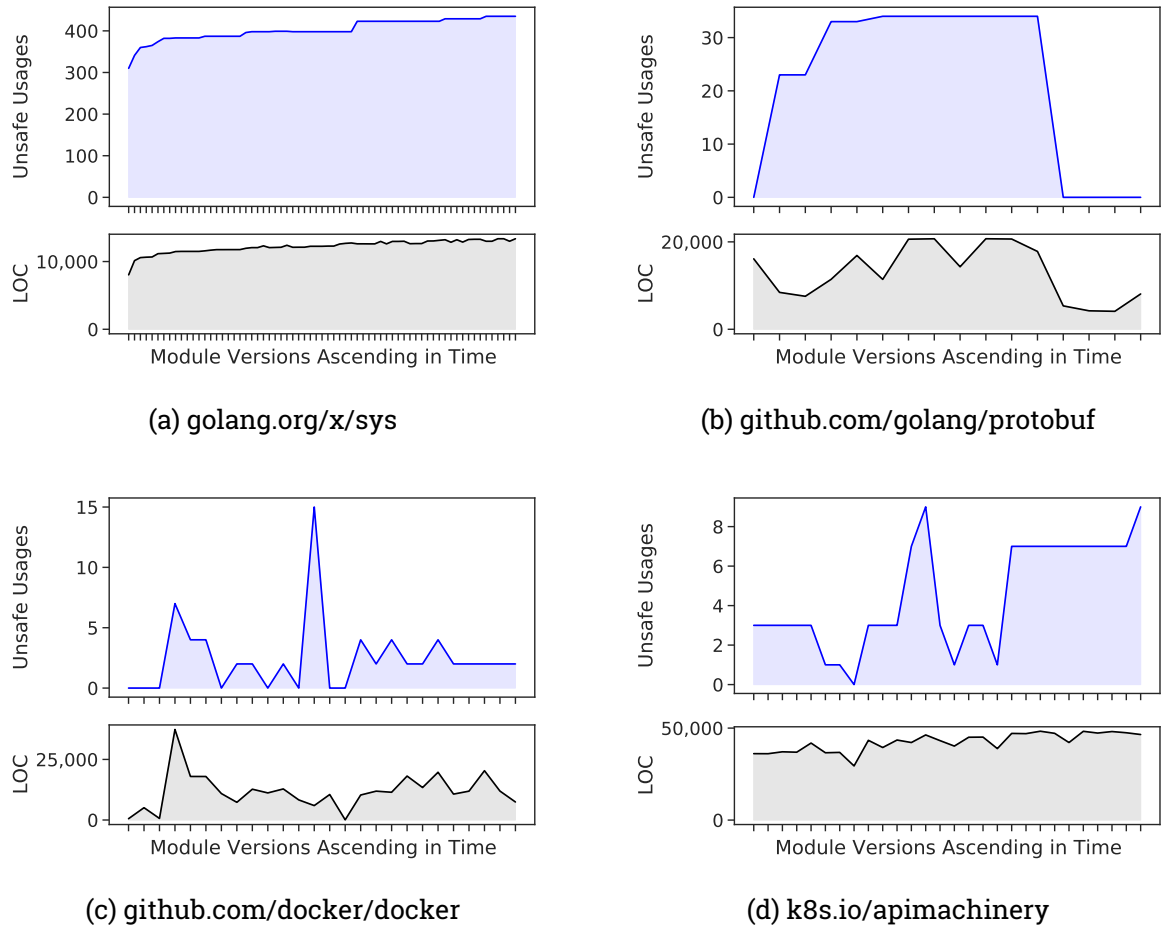
The plot shows spikes at 0 and between 1,000–2,000 usages of *unsafe*. Otherwise, the plot shows a rather uniform distribution between *unsafe* usages and the different project metrics, except for a gap between a few hundred and 1,000 usages of *unsafe*. There are both many projects with fewer and with more usages, but none with about 750. This could be an indicator that once projects have included at least a couple of *unsafe* usages, they tend to have a lot of them rather easily. There is no obvious correlation between neither *unsafe* usage and project age, nor number of stars or forks.

Answer to RQ4

There is no significant correlation between the number of *unsafe* usages and a project's age, number of stars, or number of forks.

4.3.4. Change of *Unsafe* Usage over Time (RQ5)

The data set collected in this study contains one version of each analyzed project. It is not directly possible to measure the change of *unsafe* usage in projects over time with the data available. However, there are a number of modules that are included in several versions by different projects, which means that these modules allow such an analysis of changes in *unsafe* usage. To answer research question RQ5, this subsection discusses the differences between different versions of four selected modules. These modules are *golang.org/x/sys*, *github.com/golang/protobuf*, *github.com/docker/docker*, and *k8s.io/apimachinery*.



(a) golang.org/x/sys

(b) github.com/golang/protobuf

(c) github.com/docker/docker

(d) k8s.io/apimachinery

Figure 4.7.: Change of unsafe usage in selected packages over time.
Each x tick represents one version. Note different scaling on y axes.

Figure 4.7 shows the number of *unsafe* usages contained in these modules by their respective versions in blue, ascending over time. Furthermore, the number of lines of code (LOC) is plotted

in gray to show possible correlations with changes in module size. It is important to note that the y axes in Figure 4.7 are scaled differently. This is because the purpose of that figure is not to compare the modules against each other, but to see changes in their usage of *unsafe* over time.

Depending on the specific module, there are different conclusions. For the *golang.org/x/sys* module (Figure 4.7a), it is evident that there is a monotonous increase both in *unsafe* usage and LOC. The number of *unsafe* usages increases from 315 to 440 (+39.7%) over the period shown in the figure, which is from late 2017 to May 2020 (2.5 years). A manual analysis of the changes in the module source code shows that the increased usage of *unsafe* in this case is caused by additional system call APIs that are supported by the module. Dispatching to the underlying system call code requires the use of *unsafe.Pointer*. Therefore, in this case more features provided by a dependency cause more *unsafe* code to be imported into a project.

In the *github.com/golang/protobuf* module (Figure 4.7b), there is initially a steep increase in *unsafe* usage, although the module size (LOC) is decreasing. Then, the number stays fairly constant until it drops to zero. Thus, the module does not use *unsafe* any longer, while the LOC are increasing again. It can be concluded that the module developers have replaced *unsafe* with alternative, safe features of Go.

The *github.com/docker/docker* module (Figure 4.7c) shows fluctuations in the use of *unsafe*, which are somewhat correlated to the module size (LOC). However, there is a big spike in *unsafe* usage which is not reflected in the LOC at all. Also, the number of *unsafe* usages overall is fairly low in this module. The last module, *k8s.io/apimachinery* (Figure 4.7d), is similar. Its *unsafe* usage also has a limited correlation to the LOC, but contains deviations. Here, *unsafe* has become more used in the module over time. Still, it is only a very small fraction of the module size, which has almost 50,000 LOC.

In conclusion, the usage of *unsafe* changes between different versions of the same module. Therefore, it is important for developers to consider this when updating dependencies. Furthermore, security analyses must be done for each version of a module. It is not sufficient to determine that *unsafe* usages do not enable vulnerabilities once, instead it must be checked for the specific version that is imported into an application.

Answer to RQ5

Changes in *unsafe* usage in particular modules are motivated, for example, by new API requirements and can be significant, with about a 40% increase over 2.5 years found in the *sys* module. Since there is ongoing fluctuation in the usage of *unsafe* over time for specific modules, security analyses must be done for each version individually.

4.3.5. Comparison with Existing Tools (RQ6)

To evaluate the benefit *go-geiger* provides in comparison to existing static analysis tools for Go, its findings are put in context with the results of *go vet* and *gosec* in this section. The goal of this comparison is to see whether any of those tools can achieve the same as *go-geiger* does. As described in Section 2.6, *go vet* is a linter that is included as part of the standard Go command line tool chain. It runs a number of analysis passes to identify general problems with the source code. There is the *unsafeptr* pass, which is designed to find potential misuses of the *unsafe.Pointer* type. It is however not designed for a general identification of *unsafe* usages. On the other hand, *gosec* is a static analysis tool with a design focused around security problems. It is built from several rules to identify issues, one of which (*G103*) is simply triggered by the presence of *unsafe* package members and generates a warning that those usages should be audited. Given this design, it is closer to *go-geiger* in the sense that it only identifies the presence of *unsafe* without using any logic to determine potential misuses.

To conduct the comparison, *go vet* and *gosec* are run on the same 62,025 packages that were analyzed with *go-geiger*. The results are part of the data set as well. Then, the findings of the tools are matched using package name, file name, and line number information. Table 4.1 shows the results of this analysis. The result columns are divided to show the different results for *go vet* and *gosec*. The *both* column contains the number of lines of code that were both flagged by *go-geiger* and the respective linter tool. Lines that were only flagged by *go-geiger*, but not by the linter, are shown in the following column. The last column indicates LOC that the linter flagged, but were not identified by *go-geiger*. The table contains two rows that show the number of lines of code, both when counting any message that *go vet* or *gosec* produced, and when only messages related to their *unsafe* analyses are taken into account. The latter has more impact, because comparing only those messages to the output of *go-geiger*, which is solely designed around *unsafe* usages, achieves a fairer evaluation.

Table 4.1.: Comparison of number of lines with unsafe found by go-geiger and existing linters go vet and gosec

Scenario	**both**		**only *go-geiger***		**only existing linter**	
	go vet	gosec	go vet	gosec	go vet	gosec
Any message	219	36,279	76,738	40,678	31,224	114,306
Related message	213	26,267	76,744	18,019	0	0

The results show that there is only a very small number of LOC that *go vet* and *go-geiger* found both, but many which were identified only by *go-geiger*. This means that for most of the *unsafe* usages, *go vet* does not generate a warning. When comparing any message generated by *go vet*, there are many findings that *go-geiger* does not include, however those do not exist anymore when the analysis is restricted to *go vet* messages related to *unsafe*. While *go vet* provides a lot of

warnings that are related to other problems, it does not offer any benefit over *go-geiger* for the specific task of identifying *unsafe* usages. For *gosec*, there are a lot more LOC in the *both* column, but still a lot of the *go-geiger* findings are missed. About half of the *go-geiger* results are also found by *gosec*. This is much better than *go vet*, but it is still not accurate. One reason for this is that *go-geiger* identifies not only *unsafe* package usages, but also *uintptr*, which is common as described in Section 4.3.2. It is worth noting that the numbers in Table 4.1 refer to lines of code rather than *unsafe* findings, but one line of code can contain several *unsafe* usages. Therefore the numbers do not add up to the same count of total findings discussed in Section 4.3.2.

Answer to RQ6

The existing tools *go vet* and *gosec* do not provide any benefit over *go-geiger* for the specific task it is designed for. Instead, *go-geiger* finds all and more of their *unsafe*-related results.

4.4. Qualitative Evaluation

This section presents an in-depth, qualitative study of the purpose of *unsafe* usages in ten selected open-source Go projects.

4.4.1. Labeled Data Set of *Unsafe* Usages

To study the purpose of *unsafe* in applications, a manually labeled data set of 1,400 code samples is presented as a contribution of this thesis. It is available online in the same data repository[8] as the study results presented in the previous section. The size of this data set was chosen to both allow manual labeling in a reasonable time frame, and provide a good insight into what operations are done in practice using the *unsafe* API, and for what purpose. Each sample is labeled in two dimensions by the operation type and its higher-level objective. The 1,400 code samples are drawn from the projects listed in Table 4.2. They are a subset of the projects used for the empirical study and marked by an asterisk (*) in the appendix (Table A.1). These projects were selected based on their high number of *unsafe* usages, and it was taken care of a reasonably large diversity in their domains of application. They contain applications based around virtualization, infrastructure as a service, data storage, and machine learning.

The samples are divided into 400 samples that are part of the standard library (*std*), and 1,000 non-standard-library application samples (*app*). This decision is based on the results presented in Section 4.3.2, which show that all projects must use the standard library, as well as the hypothesis that the standard library uses different *unsafe* patterns. Standard library is defined as the packages that live in the Go *std* and the *golang.org/x/sys* module. The *sys* module is included

[8] `https://github.com/stg-tud/unsafe_go_study_results`

Table 4.2.: 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`

because it contains a lot of system call infrastructure and is the replacement for the deprecated *syscall* package[9], which is part of the standard library. Thus, it is also maintained by the core Go development team. Application code samples are taken from packages that are not part of the standard library. The 1,400 code examples were sampled randomly from all packages present in the dependency trees of the projects shown in Table 4.2. However, they are taken without duplicating lines that are present in different versions of the same module. If a module is included in multiple versions, a line of code within it can not be drawn twice from different versions. Furthermore, this labeled data set contains only usages of *unsafe.Pointer*. The goal of this is to allow a better comparison between usage purposes, without distortions from different *unsafe* token types such as *Alignof* or *Offsetof*.

The classes are outlined briefly in the following. For the *unsafe* operation type dimension, Figure 4.8 shows code examples for each of the classes. The most common ones are conversions between types. The *cast-basic*, *cast-bytes*, *cast-struct*, *cast-header*, and *cast-pointer* classes represent conversions between arbitrary types, where one of them is a basic type such as *int*, a *[]byte* slice, a slice or string header structure, an actual *unsafe.Pointer*, or a structure, respectively. *Pointer-arithmetic* is a class containing any form of arithmetic address manipulation such as advancing an array. When *unsafe* is only required to pass values along to another function that expects a parameter of type *unsafe.Pointer*, the *delegate* label is applied. Thus, for this class the need for *unsafe* is at a different location in the code. The *memory-access* class is used where *unsafe.Pointer* values are dereferenced, used to manipulate corresponding memory, or for comparison with another address. *Syscall* represents calls using the Go *syscall* package or *golang.org/x/sys* module. The *definition* class denotes usages where a field or method of type *unsafe.Pointer* is declared for later usage. Finally, *unused* contains instances of *unsafe* that are not actually used in the analyzed code, such as dead code or unused parameters.

Figure 4.9 shows the possible class labels in the purpose dimension. The *efficiency* class represents

[9] `https://golang.org/pkg/syscall`

cast-basic

```
out = (*int32)(unsafe.Pointer(in))
```

cast-bytes

```
return (*(*[10]byte)(unsafe.Pointer(x)))[:]
```

cast-struct

```
out = (*runtime.Unknown)(unsafe.Pointer(in))
```

cast-header

```
hdr := &reflect.SliceHeader{
    Data: uintptr(unsafe.Pointer(&data[i])),
    Len:  42,
    Cap:  42,
}
retVal = append(retVal, *(*[]uint8)(unsafe.Pointer(hdr)))
```

cast-pointer

```
return unsafe.Pointer(ptr)
```

pointer-arithmetic

```
unaligned := uintptr(unsafe.Pointer(&value[0])) & 3
```

delegate

```
func (encoder *Encoder) Encode(ptr unsafe.Pointer) {
    encoder.UnsafeIndirect(ptr)
}
```

memory-access

```
deReferenced := *((*unsafe.Pointer)(ptr))
```

syscall

```
n, _, errno := syscall.Syscall(syscall.SYS_RECVMSG, s,
    uintptr(unsafe.Pointer(h)), uintptr(flags))
```

definition

```
type unsafeType struct {
    ptr unsafe.Pointer
}
```

unused

```
func Encode(ptr unsafe.Pointer, stream *Stream) {
    stream.WriteEmptyArray()
}
```

Figure 4.8.: Labeled data set usage classes

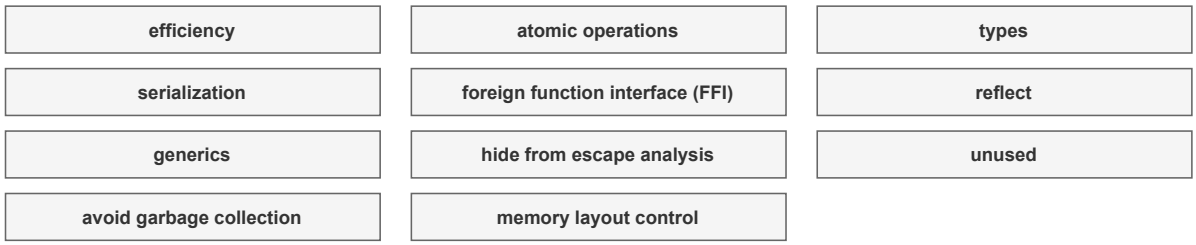


Figure 4.9.: Labeled data set purpose classes

cases where *unsafe* is used only to improve time or space efficiency of the code. Usages in this class could be rewritten to avoid *unsafe*. The *serialization* class includes (un)marshaling and (de)serialization operations, like in-place casts between complex types and bytes. *Generics* is applied where *unsafe* is used to build functionality that would be implemented using generics if they were available in current versions of Go. As of the current release plan, there is no support for generics in Go without additional libraries up until release 2.0. The *avoid garbage collection* class contains usages to tell the Go compiler to keep a value in memory while it is used, for example, when calling a function written in assembly. *Atomic operations* is a class of usages of the atomic API, which requires *unsafe* for some of its functions. The *foreign function interface* (FFI) class includes all cases of interoperability with C code (CGo), as well as calls to functions that receive their parameters as *unsafe* pointers. *Hide from escape analysis* contains instances

where *unsafe* is used to deliberately exclude a value from being seen by the EA algorithm. The *memory layout control* class represents code used for low-level memory management. *Types* samples are used by the standard library for low-level implementation of the Go type system. *Reflect* includes instances of type reflection, as well as re-implementations of types contained in the regular Go *reflect* package, for example, to use *unsafe.Pointer* instead of *uintptr* for slice headers. Lastly, *unused* is a class for unused occurrences again.

4.4.2. Purpose of *Unsafe* in Practice (RQ7)

In this section, the novel labeled data set of *unsafe* usages is used to answer RQ7 about the purpose of *unsafe* in Go applications. Table 4.3 presents the number of samples for each class. The columns denote the dimension of purpose, while the rows show the operation type dimension. Columns are divided into separate counts for the *app* and *std* groups of samples.

Table 4.3.: 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-struct	401	4	50	6	6						6	2		2		4	31						463	49
cast-basic	90	2	29	3	1						1	3			2	7	1						123	16
cast-header	36	1	3		1												3						40	4
cast-bytes	22	1	81	11							1				1		1						105	13
cast-pointer	13	8	15	13	10						16	1				2	9		1				55	33
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

The data shows that efficiency is by far the most prevalent reason to use *unsafe* in real-world Go application code. While these usages make up about 58% of the application code samples, they account for only about 5% of the standard library usages. Within the *efficiency* class, casting operations cover most of the usages with 97% (*app*) and 80% (*std*) of the samples. Next, the second most important motivation for *unsafe* code in the application class is performing serialization or deserialization operations, including marshaling of structured data to interchangeable formats. This accounts for 28% of the *app* usages. The standard library shows a different most common usage, which is avoiding garbage collection with 35%. This purpose is only found in 2% of the *app* samples. Next, the *type* (18%), *FFI* (15%), and *memory layout* (13%) classes are common in the *std* samples. A common distribution is the *hide from escape analysis* class, which is rare both in the *app* (0.1%) and *std* (2%) groups. The same is true for the *reflection* class (1% in both sets). There are also only few samples (2%) in the *app* group that are used to implement

generics functionality. Some of the findings in the *serialization* class could however be achieved with generics as well, so the classes overlap slightly.

In conclusion, it is evident that the most important reasons to use *unsafe* in Go applications are increasing efficiency through in-place conversions, and serialization with direct casts. Interestingly, both of them are achievable without *unsafe* as well, so the code could be rewritten to remove these usages. However, this could have performance drawbacks, because in-place conversions would not be possible then.

Answer to RQ7

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.

4.5. Summary

With *go-geiger*, this thesis contributes a tool to find and count *unsafe* usages in Go packages, including their dependencies. Its design allows to distinguish usages by the context they appear in, which enables filtering and, thus, a focused review of specific types such as function parameter declarations. A study on *unsafe* usage in 343 popular Go projects on *GitHub* revealed that while only about a third directly contain *unsafe*, almost all of them import *unsafe* code through their dependencies.

To study what operations are carried out using *unsafe*, and for what purpose, a novel data set of 1,400 labeled code samples was presented. It showed that the most common use cases are increasing efficiency by avoiding memory reallocation when converting values between different types, serialization and marshaling, and control over the Go memory management system when interacting with the foreign function interface (FFI).

5. *go-safer*: Detecting *Unsafe* Misuses

Another major contribution of this thesis is the development of *go-safer*, a *go vet*-style, open-source linter tool with a focus on the *unsafe* API in Go. It can identify some of the insecure code patterns described in Chapter 3. Figure 5.1 shows the organization of this chapter. It describes the design and implementation of *go-safer* based on *unsafe*-related vulnerabilities and usage data. Further, an evaluation of its effectiveness using the labeled data set of *unsafe* usages introduced in the previous chapter, a manual analysis of open-source Go packages, and a comparison with existing tools is presented.

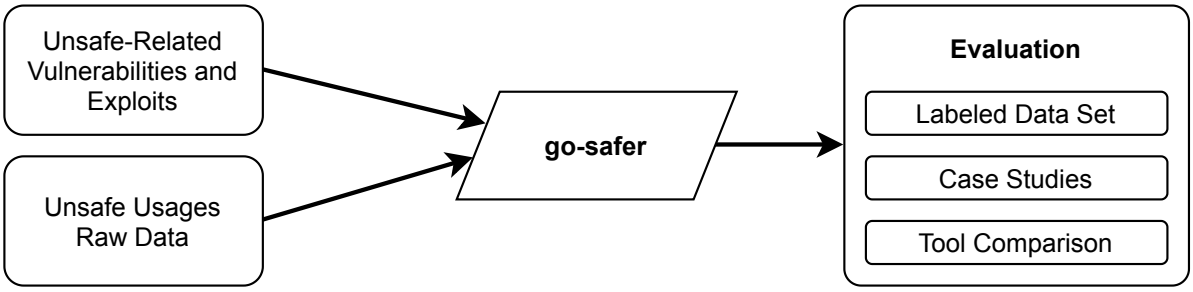


Figure 5.1.: Organization of Chapter 5

5.1. Design

The *go-safer* static analysis tool is designed as a linter to identify two misuse patterns of the *unsafe* API that were previously undetected with existing linters such as *go vet* and *gosec*. These patterns are automatically detectable using static analysis. For some vulnerabilities discussed in Chapter 3, this is not possible or too hard for the scope of this thesis. The selection of the code patterns that are detected is based on the *unsafe* usage examples identified using *go-geiger* in popular open-source Go projects as described in Section 4.4.1, as well as the manual analysis of possible vulnerabilities related to the use of the *unsafe* API presented in Chapter 3. The first is the incorrect conversion pattern between slices and strings by creating their header structures as composite literals as described in Section 3.2.2. Listing 5.1 shows a code example that uses

Listing 5.1: First vulnerable code pattern detected by go-safer

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

this pattern. The insecure creation of a *reflect.SliceHeader* instance is done in Lines 3–7.

In addition to composite literals of type *reflect.SliceHeader* and *reflect.StringHeader*, *go-safer* analyzes accesses to fields of existing instances of these types, unless they are derived by cast from a real slice or string. If they are, then their *Data* field is treated as a reference by the Go runtime, as described in Section 2.2. Furthermore, if the slice or string header types are renamed to a different type name in the source code, field accesses and composite literals of the new type name are recognized just like the original types.

The second misuse that *go-safer* detects is a direct conversion between struct types containing incompatible types with architecture-dependent sizes or alignments as described in Section 3.1. Listing 5.2 shows an example of such code. Here, Line 8 contains an incorrect cast between the incompatible structures *A* (Lines 1–3) and *B* (Lines 4–6).

Listing 5.2: Second vulnerable code pattern detected by go-safer

```
1 type A struct {
2    x int
3 }
4 type B struct {
5    y int64
6 }
7 func unsafeFunction(a A) B {
8    return *(*B)(unsafe.Pointer(&a))
9 }
```

The source code and documentation of *go-safer* is available on *GitHub*[1]. Figure 5.2 shows a screenshot of *go-safer*. The output contains a few warnings about insecure slice conversions, which are detected in the *nil_cast* package that is part of the *go-safer* test suite. For each warning, one line is generated in the output including the source file, line number, and column of the insecure code.

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

```
/tmp/go-safer $ go-safer ./passes/sliceheader/testdata/src/bad/nil_cast
/tmp/go-safer/passes/sliceheader/testdata/src/bad/nil_cast/nil_cast.go:12:2: assigning to incorrectly derived reflect header object
/tmp/go-safer/passes/sliceheader/testdata/src/bad/nil_cast/nil_cast.go:13:2: assigning to incorrectly derived reflect header object
/tmp/go-safer/passes/sliceheader/testdata/src/bad/nil_cast/nil_cast.go:14:2: assigning to incorrectly derived reflect header object
```

Figure 5.2.: Usage example screenshot of go-safer

5.2. Implementation

The *go-safer* tool is built using the Go analysis infrastructure that is available as part of *go vet*. This allows to compose analysis steps as modular and reusable parts, which are called *passes*. Each analysis pass is run as a unit on the source code packages under analysis. It can depend on the results of other analysis passes, which are run before and hand the results over to it. The passes and their relations must form a directed acyclic graph, and the analysis infrastructure code determines the optimal execution order and concurrency. There are two novel analysis passes in *go-safer*, the *sliceheader* and the *structcast* pass. Figure 5.3 shows an overview of this architecture.

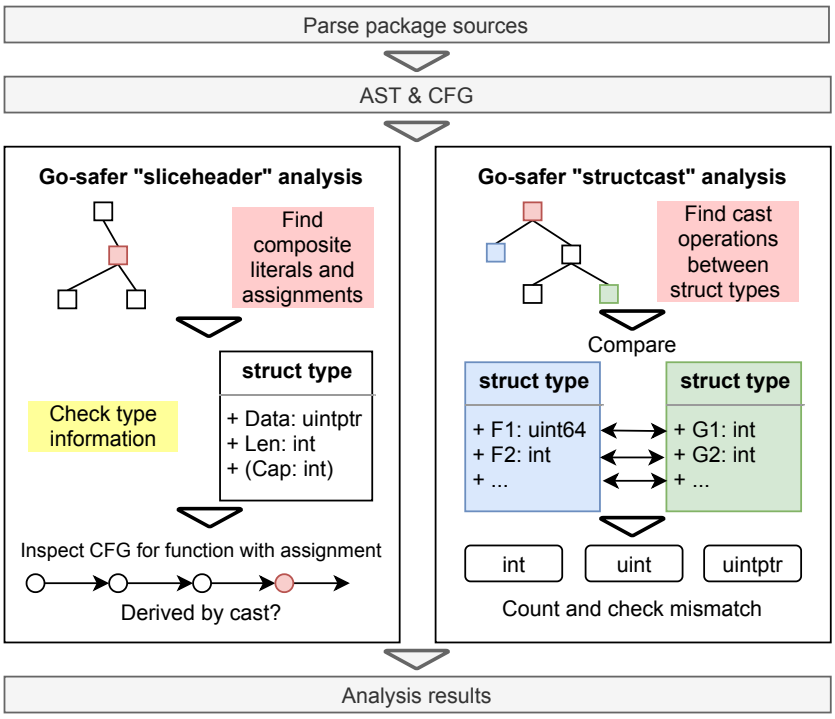


Figure 5.3.: Architecture of the go-safer static code analysis tool

Parts that are shown in gray in Figure 5.3 are reusing existing code. Parsing the source code of the packages under analysis is done in the same way that *go vet* does, in fact that code is imported

as a dependency for *go-safer*. Then, the abstract syntax tree (AST) and control flow graph (CFG) of the sources are built using existing analyses, which the novel *go-safer* passes depend upon. After all analyses are finished, the results are presented to the user. This composition using many existing parts allows to keep the required new code to a minimum, reuses well-tested pieces of code, and allows integrating the new *go-safer* features into a familiar workflow.

The *sliceheader* analysis pass finds the code pattern shown in Listing 5.1. First, it receives the AST and filters it for composite literals (*CompositeLit* nodes) and assignment statements (*AssignStmt* nodes). Recursively going down the AST even after a composite literal has been found ensures that *go-safer* also detects literals that are part of a bigger structure type that contains a slice header as one of its fields. Then, the type of either the composite literal or the assignment receiver is checked. For assignments, this is done by looking up the left hand side identifier in a type table provided by the parser. Since Go supports multiple assignments in one statement, there can be multiple variables on the left hand side. In this case, *go-safer* checks all of them to see if any meet the conditions to trigger a warning. To check whether the type matches either *reflect.StringHeader* or *reflect.SliceHeader*, it is checked for assignability to the underlying structure of the header types. This structure, as illustrated in Figure 5.3, is one *uintptr* field, and one or two *int* fields. Comparing to the structure instead of the name of the type makes it possible to detect type aliases of the header types which might exist. An example of this is shown in Listing 5.3. It contains a type definition for *CustomHeader* (Line 1), which is then used to create a composite literal (Lines 4–8).

Listing 5.3: SliceHeader alias type detected by go-safer

```
1  type MyHeader reflect.SliceHeader
2
3  func unsafeFunction(address uintptr) []byte {
4     header := &MyHeader{
5        Data: address,
6        Len: 42,
7        Cap: 42,
8     }
9     return *(*[]byte)(unsafe.Pointer(header))
10 }
```

A composite literal of a header type is enough to issue a warning, but for assignments, *go-safer* needs to check whether the receiver variable is a slice or string header that was created by conversion from a real slice or string. To do this, the AST assignment node is identified within the CFG. Then, going backwards in the CFG, the last node that assigns the value of the receiver variable is found. This node defines the effective value of the slice or string header structure at the time of the assignment under investigation. Finally, *go-safer* checks whether the right hand side value of the statement defining that value is a conversion from an actual slice or string value. It has to be a cast through an *unsafe.Pointer* value for this to be possible. If that is the case, the assignment is legitimate and *go-safer* does not generate a warning. However, if the variable

is not derived by a conversion from a slice or string, or no CFG node defining the variable can be found, a warning is issued. This is possible, for example, if the slice or string header is passed as a function parameter, because its value can not be statically inferred with the Go analysis framework in that case.

The *structcast* pass is responsible for detecting the code pattern shown in Listing 5.2. It also uses the AST to find cast operations. These are represented in the AST in the same way as function calls are, so the tree is filtered for *CallExpr* nodes. If there is a cast from some structure type to *unsafe.Pointer* and further to another structure type, these source and target types are analyzed. These types can again be fetched from a type lookup table provided by the parser. For both of them, *go-safer* counts the number of fields that have the types *int*, *uint*, or *uintptr*. These three basic types are the ones available in Go that have different sizes, alignments, and byte orders on varying architectures. For example, *int* is eight bytes on the *amd64* architecture, but only four bytes on *i386*. If the counts mismatch, the direct cast using the *unsafe* API will likely break on some architectures, and *go-safer* generates a warning.

5.3. Evaluation

In order to make a valuable contribution to the security of Go programs, and also be an effective tool for developers, it is important that *go-safer* can find actual bugs in real-world Go projects, and at the same time have a low rate of false positives and false negatives. False negatives would mean that the tool misses incorrect and therefore potentially vulnerable usages of the *unsafe* API. On the other hand, a high rate of false positives would mean that it generates many warnings that are not related to any programming errors [4]. This would require developers to manually review the findings and dismiss many of them, which decreases the value of *go-safer* significantly. First, developers could lose trust in the tool and refuse using it with their regular development workflows. Second, false positives make it hard to integrate *go-safer* with automated continuous integration processes, because the build would fail without there being a real problem.

To demonstrate that *go-safer* is capable of finding actual bugs while having an acceptable rate of both false positives and negatives, its performance is evaluated in two different ways. One of them uses the novel data set of labeled *unsafe* usages, which was presented in Section 4.4.1. The second one is done by manually analyzing six selected open-source Go packages from different projects and comparing the results to the output of *go-safer*. Finally, the performance is compared to the existing static analysis tools *go vet* and *gosec*.

In addition to this academic evaluation, *go-safer* was also used to discover 64 bugs that existed in the open-source projects used for the study on real-world usage of *unsafe* presented in Chapter 4, or their dependencies. By the time this thesis was submitted, the authors of the affected projects have acknowledged 59 (92%) of those bugs, and accepted the fixes that were submitted to them.

5.3.1. Labeled Usages from the Data Set

To evaluate the performance of *go-safer* using the novel data set of labeled *unsafe* usages, first all instances from the *cast-header* class are taken from the set. These are possible misuses involving incorrect constructions of slice and string headers. There are 44 such code samples, which are manually classified as positive and negative examples. Positive means that the code is a misuse of the *unsafe* API, and *go-safer* should generate a warning for it, while negative means that the code is a correct and safe usage. There are 30 positive and 14 negative samples.

Then, *go-safer* is run on all packages that contain the respective 44 *unsafe* usage samples, and all warnings issued are saved into a CSV file. This makes it easy to match the warnings with the labeled samples by their file name and line number. Samples that are classified as positive are counted as true positives (TP) if *go-safer* generates a warning for it, and false negative (FN) otherwise. Similarly, negative samples count as false positives (FP) if there is a warning, and true negatives (TN) otherwise. Table 5.1 shows the results of this evaluation in its first row. The other rows contain results from other linters and are discussed in Section 5.3.3. Furthermore, the table contains the resulting precision, recall, accuracy, and F1-score as calculated from the classes counts.

Table 5.1.: Evaluation results for go-safer on the labeled data set of unsafe usages

Tool	TP	FP	TN	FN	Precision	Recall	Accuracy	F1-Score
go-safer	29	1	13	1	0.967	0.967	0.955	0.967
go vet	0	0	14	30	-	0	0.318	0
gosec	29	13	1	1	0.690	0.967	0.681	0.805

The results show both high precision and recall of 96.7% for *go-safer*. High precision means that there are few false positives, so the warnings generated by *go-safer* are almost always correct and show evidence of an actual bug in the code. Therefore, developers can include it into their workflows without the need of dismissing many invalid warnings. A high recall shows that there are few false negatives, therefore a high fraction of the existing bugs is detected by *go-safer*. Thus, developers have a reliable tool at hand to be trusted in detecting bugs of the particular class of *unsafe* misuses that it is specialized in. Accuracy and F1-score are high as well, which follows from both high precision and recall and emphasizes the quality *go-safer* provides.

Evaluation results on labeled usages

Evaluated using manually labeled *unsafe* usages, *go-safer* achieves both 96.7% precision and recall. Therefore, its accuracy is 95.5%.

A manual analysis shows that the one false positive is due to a slice header that is created from a

real slice, however the slice is referenced through a slightly obfuscated, reflected pointer which *go-safer* can not connect to the slice. The false negative is caused by compilation errors in the package which prevent *go-safer* from analyzing the package altogether. Since it depends on the AST, its static analysis passes can not run and the execution stops with a generic error message before the parsing step has finished. The same is true for *gosec*. To work around this scenario, *go-safer* would need a secondary analysis strategy that works on the source code instead of the AST, which is not feasible with the tools presented in this thesis.

5.3.2. Case Study Packages

A second evaluation of *go-safer* is done by manually analyzing six selected Go packages from open-source projects. These packages are chosen using the results of the study on *unsafe* usage in open-source code presented in Section 4.3. Evaluating *go-safer*, an *unsafe*-related linter, on packages that do not actually contain any *unsafe* usages does not provide valuable insights. Table 5.2 shows the six packages used for this evaluation, including their number of Go files, lines of code (LOC), and *unsafe* usages. They are manually selected from the packages in the dependency trees of the projects analyzed with *go-geiger*. To create a variation both of the number of *unsafe* usages and package size, they are chosen to include two packages each with few, medium, and many *unsafe* usages. For every category, there is one small and one large package in terms of LOC each.

Table 5.2.: Selected packages for the evaluation of go-safer

Package	Number of Go Files	LOC	Unsafe Usages
k8s.io/kubernetes/pkg/apis/core/v1	6	10,048	677
gorgonia.org/tensor/native	4	1,867	158
github.com/anacrolix/mmsg/socket	86	3,782	115
github.com/cilium/ebpf	14	2,851	65
golang.org/x/tools/internal/event/label	1	213	8
github.com/mailru/easyjson/jlexer	4	1,234	5
Total	**115**	**19,995**	**1,028**

The six packages were analyzed manually to find any insecure code that *go-safer* should detect. The results of this were saved as a CSV file containing a positive or negative label for every line of code that includes a usage of the *unsafe* API. Lines that do not use the API should not be analyzed by *go-safer* at all, therefore they are not included as negative in order to avoid a very large number of true negatives which would skew the accuracy and yield less valuable evaluation results. Should *go-safer* generate a warning for such a line, it will still be counted as false positive. After the manual analysis, *go-safer* was run on the packages. Similar to Section 5.3.1, the warnings and labels were matched and counted.

Table 5.3.: Evaluation results for go-safer on manually analyzed packages

Tools: a go-safer, b go vet, c gosec

Package	TP			FP			TN			FN			Precision			Recall			Accuracy			F1-Score		
	a	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*
v1	0	0	0	0	0	676	677	677	1	0	0	0	-	-	0	-	-	-	1	1	0.001	-	-	-
native	48	0	0	9	0	98	101	110	12	0	48	48	0.842	-	0	1	0	0	0.943	0.696	0.076	0.914	-	-
socket	0	0	0	0	0	16	115	115	99	0	0	0	-	-	0	-	-	-	1	1	0.861	-	-	-
ebpf	0	0	0	1	0	38	64	65	27	0	0	0	0	-	0	-	-	-	0.985	1	0.415	-	-	-
label	0	0	0	0	0	7	8	8	1	0	0	0	-	-	0	-	-	-	1	1	0.125	-	-	-
jlexer	1	0	0	0	0	2	4	4	2	0	1	1	1	-	0	1	0	0	1	0.8	0.4	1	-	-
Total	**49**	**0**	**0**	**10**	**0**	**837**	**969**	**979**	**142**	**0**	**49**	**49**	**0.831**	**-**	**0**	**1**	**0**	**0**	**0.990**	**0.952**	**0.138**	**0.907**	**-**	**-**

Table 5.3 shows the results of the evaluation for each package individually as well as in total. The numbers for *go-safer* are in the first column for each value, while the other two columns are discussed in the following section for a comparison to existing linters again. In this second evaluation, *go-safer* demonstrates a slightly lower precision of 83.1%, which means that about one in six warnings is a false positive. The recall however is at 100%, still resulting in an excellent accuracy of 99% and F1-score of 90.7%.

The false positives generated by *go-safer* for the *native* package are caused by the same reason as the false positive described in the previous section. A slice header is created from a real slice, but the slice is referenced through an obfuscated pointer. However, this time all of the occurrences of that pattern in the package are found. For the *ebpf* package, the false positive is an exact match of the detection rules implemented by *go-safer*, however in that particular case the usage is not vulnerable to the possible *use-after-free* bug described before, because a call to the *runtime.KeepAlive* function prevents the underlying data array referenced in the created slice header from being freed when the garbage collector runs. Currently, *go-safer* does not detect this, which is a limitation of its rule set.

Evaluation results on case study packages

Evaluated using six selected packages, *go-safer* is able to score an accuracy of 99%, with a precision of 83.1% and recall of 100%.

5.3.3. Comparison with Existing Tools

In this section, the performance of *go-safer* is compared to the previously existing static analysis tools *go vet* and *gosec*. For both evaluation sets described in the previous sections, these tools were also run and their respective generated warnings were saved. This allows to identify true and false positives and negatives in the same way as for *go-safer*. Only the respective warnings that are related to the *unsafe* API are counted. For *go vet*, this is the "*possible misuse of unsafe.Pointer*" message generated by the *unsafeptr* pass, and for *gosec* it is rule *G103* resulting in the "*use of*

unsafe should be audited" message.

The results are presented in Tables 5.1 and 5.3 next to those of *go-safer*, in the remaining two rows and columns *b* and *c*, respectively. Both evaluations, on the labeled data set and six manually analyzed packages, reveal that *go vet* did not generate any *unsafe*-related warnings, while *gosec* simply flagged all usages of *unsafe* as potentially dangerous. The *gosec* results contain some false negatives, because there are lines of code in the evaluation data set that do not contain a direct call of a function belonging to the *unsafe* package. These are assignments to slice and string headers that are previously created, which are detected by *go-safer* but not *gosec*. Similarly, there are no true positives for *gosec* in Table 5.3, because there are no positive samples in that evaluation set that contain a direct usage of *unsafe.Pointer* in the offending line of code.

The reported performance metrics are not overly meaningful for *go vet* and *gosec*. With the evaluation on the labeled data set of *unsafe* usages shown in Table 5.1, *go vet* has a low accuracy of 31.8% although precision and recall are not defined. This accuracy simply mirrors the fraction of negative samples in the overall sample set. Similarly, the inverse of that fraction (68.1%) is seen as the accuracy of *gosec* in that table. It also has a rather high precision and recall but this is simply the result of generating warnings for every sample in the set. In the second evaluation on the six packages, none of them achieves a defined and greater-than-zero precision or recall for either *go vet* or *gosec*, because both tools do not report any true positives. However, *go vet* still gets a high accuracy of 95.2%, because this evaluation set contains a lot more negative samples than positive ones.

In summary, this comparison shows that neither *go vet* nor *gosec* are suitable for the purpose that *go-safer* fulfills, the tools are focused on different details of the same overall goal of making Go applications safer.

5.4. Summary

Existing tools like *go vet* and *gosec* miss several insecure usage patterns of *unsafe*. In this chapter, *go-safer* was presented. It is a novel linter for Go code that can detect insecure constructions of *reflect.SliceHeader* and *StringHeader*, and conversions between architecture-dependent types which can become invalid on different platforms. With these checks, the vulnerabilities described in Sections 3.1, 3.2.2 and 3.2.3 are prevented. Since *go-safer* is based on the analysis framework used by *go vet*, it could easily be integrated into it in the future.

An evaluation of *go-safer* showed that it can achieve more than 95% accuracy. Its high precision means that there are few false positives, which would need to be manually reviewed and dismissed. Thus, it is a valuable tool for Go developers and security analysts.

6. Related Work

This chapter discusses related and concurrent work relevant to this thesis. First, a similar, concurrent study on *unsafe* code usages in the Go programming language is presented, replicated, and compared in detail to this work. Then, related studies and papers are discussed in groups identified by their general topic.

6.1. Concurrent Study on *Unsafe* in Go

Costa et al. submitted a similar study to this work on usages of the *unsafe* API in open-source Go code to the IEEE Transactions on Software Engineering journal. At the time this thesis is written, their paper is not yet accepted, thus, it is cited from arXiv [10]. While I personally could not find substantial errors in the paper, it has not yet been peer-reviewed. Since it is a very relevant concurrent work to this thesis, I nevertheless reproduced their empirical study and compared it to the results of this thesis.

The authors present a study of 2,438 popular Go projects, which were downloaded on October 2, 2019. Their project selection was done by taking the top 3,000 most-starred open-source projects on *GitHub*, filtering out archived, inactive, and young projects with less than 10 commits. Furthermore, they removed educational projects such as books or learning material, as well as projects that could not be downloaded for some reason. Then, they counted the number of *unsafe* usages in each project using a parser to generate the abstract syntax tree (AST). They found that 24% of the studied projects use *unsafe* code. In contrast, the study presented in Chapter 4 in this thesis finds that 38.19% use *unsafe*. This difference is partly due to the different choice of projects, which was verified by comparing the set of analyzed projects available through the published replication package[1]. While Costa et al. analyzed more than 2,400 projects, this thesis only analyzed 343, indicating that *unsafe* usage is more prevalent in more popular projects. Furthermore, a fundamental difference between this work and [10] is that the study by Costa et al. did not look at dependencies, instead they only analyzed the projects directly. My analysis on the projects including their dependencies in contrast showed that 90.96% of the projects use *unsafe* either directly or through some included library. To compare the study to [10], it

[1] `https://zenodo.org/record/3871931`

is therefore necessary to focus on the *unsafe* usages contained in the top-level projects, which is done by taking into account only usages that are detected within the main project module indicated by the *go.mod* file in the root directory. However, this can be inaccurate if there are multiple modules contained within the same project repository, because in that case the study by Costa et al. would attribute code directly to a project, but it is seen as a dependency in my study. There are 25 projects for which [10] report *unsafe* usages but my study does not. However, they are all included in the 90.96% of projects for which my study indicates that they use *unsafe* code through their dependencies. For this reason, it can be concluded that the difference in the fraction of projects using *unsafe* is due to a different definition of project and dependency code.

There are 86 projects for which the study in this thesis and [10] report different absolute numbers of *unsafe* usages, while the numbers match exactly for all remaining projects. To find the reason for the difference, their study was replicated in the best possible way, given that Costa et al. did neither include the exact revisions of the projects under analysis nor their parsing tool to count the *unsafe* usages in their replication package. Table 6.1 shows the top 10 projects with highest difference, including their respective absolute counts found in the studies and the reason for the difference.

Table 6.1.: Comparison of unsafe usage counts in Costa et al. and this work
Reasons: a change over time, b multiple architectures, c dependency vendoring, d submodules

Project	Count in [10]	Count in this work	Difference	Reason
jetstack/cert-manager	140	374	+ 234	a
kubernetes/kubernetes	2,058	1,885	– 173	b
golang/mobile	211	93	– 118	b
TykTechnologies/tyk	79	187	+ 108	c
elastic/beats	164	70	– 94	c
golang/tools	113	44	– 69	b
peterq/pan-light	64	0	– 64	d
cilium/cilium	290	345	– 55	c
go-delve/delve	72	20	– 52	b
ethereum/go-ethereum	42	91	+ 49	c

For the *jetstack/cert-manager* project, the difference is due to the changes in the code between October 2019 and May 2020, which is when the project data set for this thesis was collected. This is indicated as *Reason a* in the table. For *peterq/pan-light*, there are multiple Go modules included in the project repository, so for this project the difference is due to varying definitions of dependency code as described previously. *Reason d* corresponds to this situation. Finally, there are four projects that did not yet support the Go module system in October 2019 and included vendored copies of their dependencies in the repository (*Reason c*), and four projects which contained separate specific code files for different architectures, which are not counted by

go-geiger and thus are not included in this work's study (*Reason b*). Because of this, the counts associated with *Reason b* are higher in [10]. The only significant methodical difference therefore is the different counting of code that is available separately for different architectures, which is a limitation of the *go-geiger* tool.

By looking at *unsafe* usage counts for each week since 2015 in the history of 270 projects selected from the total projects data set, Costa et al. showed that while the share of packages containing *unsafe* code did not change significantly, the absolute number of individual *unsafe* usages increased over time. The study by Costa et al. also includes a set of manually labeled purposes of *unsafe* usages, however the authors labeled one entire file from each project with a single label, while the labeled data set presented in this thesis is built with a granularity of a pair of labels for a single line of code. Comparing the labels was possible on only 25 code samples that are included in both sets. Furthermore, the data set presented in [10] is incomplete, or at least the version published as replication package is. In the data set by [10], 20 of the 25 mutual code samples did not actually have a label. The remaining five matched reasonably well at least, as is shown in Table 6.2. The table contains both the source code snippet and the labels assigned by [10] and this work, as well as their concordance in the third column.

Table 6.2.: Comparison of unsafe usage labels in Costa et al. and this work

Source code **Labels in [10]**	**My labels**	**Concordance**
**(*uintptr)(unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) + uintptr(i))) ^= kw*		
Performance Optimization	pointer-arithmetic / efficiency	match
return Pointer{unsafe.Pointer(v.Pointer()), v.Type()}		
Reflect	cast-pointer / reflect	match
*raw := (*unix.InotifyEvent)(unsafe.Pointer(&buf[offset]))*		
System Call	cast-struct / serialization	mine more fine-grained
*entryHdr := (*entryHdr)(unsafe.Pointer(&entryHdrBuf[0]))*		
Convert from byte	cast-bytes / efficiency	match
if n := int(uintptr(unsafe.Pointer(&b[0]))) % wordSize; n != 0 {		
Performance Optimization	pointer-arithmetic / serialization	mine more fine-grained

Three of the labels were exact matches of the label provided by [10] and one of the two labels assigned to the sample in the data set presented in this thesis. The second label is lost, because [10] only did a one-dimensional labeling. For the remaining two samples, the labels did not match, however my labels are more fine-grained with respect to the specific line of code. The labels assigned by [10] are appropriate for their scope, which is the complete file containing the line of code shown in the table. Overall, the usage purposes identified by [10] match the ones identified in this thesis with most of them being used for the foreign function interface (FFI) and efficient type casting, and few of them to achieve reflection and direct pointer manipulations.

Additional contributions of [10] include a manual correlation of *unsafe* usages to project domains, which showed that networking projects are the biggest group, followed by development tools, container virtualization, databases, and projects that offer bindings to other libraries or applications. Bindings and networking projects include the most projects with heavy use of *unsafe* (more than 100 calls in a project).

6.2. *Unsafe* APIs Across Languages

Similar to the Go *unsafe* API, there are other programming languages that offer ways to circumvent their respective measures for memory safety. Rust offers a sophisticated concept of value ownership that prevents invalid memory accesses like *use-after-free* bugs [38]. To allow developers to circumvent this safety measure when necessary, for example, to implement low-level functions, it provides the *unsafe* keyword to mark a function or code block that is allowed to do five additional, potentially unsafe operations. They include dereferencing raw pointers, calling *unsafe* functions, and accessing mutable static variables [38]. This achieves the same level of possibilities as offered by the *unsafe* API in Go. Recently, two studies have analyzed the usage of *unsafe* code blocks in open-source Rust libraries and applications. Evans et al. [16] presented an empirical study revealing that less than 30% of analyzed libraries directly contained *unsafe* code. However more than half did when their dependencies were included in the analysis. Over the course of their ten months' study, they found that these numbers did not change significantly. Most of the *unsafe* Rust code is used to call other Rust functions, while interoperability with external C code was a smaller concern. The authors conducted an N=20 survey about the reasons to use *unsafe* on Reddit, which showed that the most popular reasons were performance optimizations, advanced data structures, or a more elegant interface made possible by using *unsafe* APIs. About 10% of developers indicated that they have used *unsafe* just to make the code compile in the past.

Qin et al. [43] studied bug reports that were related to *unsafe* code usage in Rust, which revealed that the most common bug classes are buffer overflows, null-pointer dereferencing, reading uninitialized memory, and *use-after-free* bugs. Often, the cause of the bug was an incorrect check for specific edge cases, which could be fixed by conditionally skipping the execution of *unsafe* code. The authors note that it is very dangerous to have hidden *unsafe* code in regular functions, which can be called from safe Rust code, because developers often need to make sure that the input data is in a proper state, which is not immediately obvious.

Furthermore, Almohri et al. [1] presented a system to ensure memory safety while executing *unsafe* code in Rust by limiting access to the program's memory during the times *unsafe* functions run. This is done by splitting the memory address space into regions with different trust levels and, thus, creates a sandbox which the *unsafe* code runs in. RustBelt [27] is a formal proof of safety-related properties of a subset of the Rust language with *unsafe*. Since it is possible

to compile Rust to WebAssembly, a binary intermediate code representation that is shipped to web browsers [45], usage of *unsafe* blocks in Rust might lead to vulnerabilities that occur in the virtual machine environment when executing the WebAssembly code. Lehmann et al. [34] presented a study on this possibility.

Java also contains an *unsafe* API with the *sun.misc.Unsafe* library, which can cause security vulnerabilities when misused [36]. Mastrangelo et al. [37] showed that 25% of the artifacts analyzed in an empirical study used this *Unsafe* library. Huang et al. [25] analyzed the causes and consequences of misuses of *Unsafe*, and showed different patterns in which affected programs can crash due to such programming errors.

Finally, for the non-memory-safe languages C and C++, there is previous work on achieving partial memory safety at least [7, 41]. Song et al. [49] presented tools that help with the process of identifying vulnerabilities in applications. Furthermore, there have been comprehensive studies in the past about vulnerabilities related to memory safety [50, 2, 32].

6.3. Go Vulnerabilities Unrelated to *Unsafe*

There has been a lot of previous research on security vulnerabilities in Go programs that are not related to the *unsafe* API [58, 24, 22, 8]. One of the features Go is most known for is its excellent support for concurrent program execution [14]. To synchronize concurrent threads, it offers both message passing through channels and coordinating memory access by using mutually exclusive locking of variables through mutexes. Tu et al. [51] presented a study of about 170 concurrency-related bugs in six open-source Go applications. To do this, they identified commits that are related to fixing such bugs by searching the project history for related keywords. They found that there is an even distribution between bugs related to message passing and shared memory, and that the usage of both techniques did not change significantly over time. They conclude that adding new features to make a language safer is not necessarily sufficient, because new bug classes can be introduced, especially if developers are not familiar with the new concepts.

Other related work on concurrency in Go includes a detailed analysis of message passing in open-source projects [13], which found that most projects use channels for synchronous message passing, and that common models to organize concurrent threads are poorly supported by static analysis tools. Lange et al. [31] presented work on detecting dead locks and infinite loops that are caused by incorrect usages of channels. To do this, they built a model of possible communication patterns in Go applications, and applied bounded verification to find incorrect ones. Giunti [20] contributed a framework that can generate concurrent Go code that is free from data races and deadlocks. This is achieved using a formal communication model expressed in a special calculus.

Wang et al. [53] discussed how the escape analysis of the Go compiler misses the connection

between the underlying data arrays of slices and strings when they are incorrectly converted into each other by creating a composite literal header value as described in Section 3.2.3. The authors, however, used this property to optimize the heap memory consumption of Go programs. They presented a transpiler that modifies the intermediate binary representation created by the Go compiler, which is converted to architecture-dependent specific assembly afterwards. When a local variable is passed by reference to another function, it is often seen as escaped, because the Go compiler uses a conservative approach to detect escaped values. Therefore the variable will be allocated on the heap. The transpiler checks whether there is any possible concurrent access to that variable, and if there is none, it might be possible to allocate it on the stack of the calling function instead. To achieve this, the reference passed to the function is converted into a *uintptr* value and back to a reference to break the connection between the values as seen by the escape analysis on purpose. Doing this can improve heap usage, because some values are not unnecessarily allocated there.

6.4. Static and Dynamic Code Analysis

Previous work presented static code analysis tools that are designed with a focus on security vulnerabilities in mind, similar to the *go-safer* linter introduced in this work. For Android projects built in Java, there is *cryptolint* [15]. It is a static analysis tool that detects misuses of cryptographic APIs in Android apps, such as the use of the electronic code book (ECB) cipher mode. Its authors Egele et al. found that 88% of the 11,748 Android applications that they analyzed contained at least one misuse. The specification language *CrySL* [29] can be used to formally describe rules that ensure cryptographic APIs are used correctly. Static analysis tools like *CogniCryptSAST* [28] can check whether those rules are followed by an application and, thus, find cryptographic misuses that are potential security vulnerabilities. Wickert et al. [56] presented a labeled data set of 201 cryptographic misuses in Java projects. It is organized in two dimensions: the specific cryptographic API that is used, and the category of misuse. The authors integrated their novel data set into *MuBench* [3], which is both a collection of API misuses as well as a framework to evaluate precision and recall of static analysis tools on the data sets.

For JavaScript, Gong et al. [21] presented *DLint*, which is both a static and dynamic linter. On top of common problems with JavaScript source code that are also found by previous static-only linters, it adds a dynamic analysis approach that is able to check 28 additional potential problems with the code. In an empirical study on more than 200 popular websites, the authors found on average 49 problems per site that were missed with static analysis.

Song et al. [49] provided a detailed overview of available static and dynamic analysis tools for the C and C++ languages, including a taxonomy of the tools and security vulnerabilities detected by them. They find that although there has been decades of research into analysis tools for these languages, there is still room for improvement, in particular with respect to incorrect usages

of pointer types and security vulnerabilities that are caused by them. These include, among others, the detection of dangling pointers, bounds checking of buffer accesses using pointers, and finding invalid conversions between incompatible types. For the Go programming language, Bodden et al. [5] presented an analysis of information flow and possible data leaks through taint analysis. Gabet et al. [18] contributed work on the static detection of race conditions and safe access of shared memory through mutexes. Similar to *CrySL*, it is based on a formal model of communication in Go, and rules to detect incorrect instances.

Smith et al. [47] conducted a study on the usability of static analysis tools with a focus on security problems. They found that often problems that could have been found by the tools are not fixed by the developers, because the user interfaces are too complex and the tool gets abandoned. The authors recommend four design guidelines to improve the usability of static analysis tools. They include suggestions on how to fix a specific bug rather than just pointing out its existence, integrating the tools within the existing workflows of developers, showing results within the code editor near the relevant code, and contextualizing warning messages to the concrete incorrect code, such as including specific variable names.

6.5. Security Issues with Dependencies

Concerning the dependencies or imports of projects, previous work has analyzed different aspects. Pashchenko et al. [42] discussed how to focus on the most important dependencies when looking for security vulnerabilities. The study was done for Java, specifically for the Maven dependency management system. A central point is to distinguish the different scopes for testing and production, as vulnerabilities that exist in dependencies that are only relevant for testing have less impact. This is because they are not shipped to customers as part of the final product, and therefore do not run on actual production systems. The authors did not claim that security vulnerabilities in such libraries are unimportant, but when companies have a limited budget of development resources to spend on auditing dependencies, it is a good choice to focus on the libraries with the most customer-facing impact. The study found that in the analyzed Java projects, about 20% of dependencies are required for development and testing purposes only. The Go module system, which compared to Maven is very new and only recently reached a stable version, does not have a distinction between development and production dependencies. Therefore, it is not possible to account for this difference in the study presented in this thesis. Pashchenko et al. also proposed an algorithm for deciding whether a dependency is still actively maintained, or has been abandoned. They found few vulnerabilities in libraries that are no longer maintained, however those have a big impact, because there are no updates to fix them. Therefore, it is important to replace such dependencies with actively supported alternatives.

A study by Watanabe et al. [54] analyzed the origins of vulnerabilities in mobile applications available for Android. They found that 70% of them were introduced through vulnerable

dependencies. Furthermore, 50% of the bugs came from third-party dependencies, where developers can not easily fix them, and instead have to rely on the maintainers or replace the library with an alternative one. Xia et al. [57] also presented work on the prevalence of outdated libraries in open-source project dependencies across different languages.

When insecure dependencies have been identified, it is important to update or replace the affected libraries as soon as possible. Kula et al. [30] conducted a study on *GitHub* projects, as well as a developer survey, to find out how the update process is done in practice. They found that developers often think maintaining up-to-date versions for their set of dependencies is extra work that can be deprioritized. Thus, dependencies are not updated in a timely manner. Specifically, the authors found that developers are often unaware of security vulnerabilities in dependencies. As soon as they find out about them, dependencies are however usually updated quickly. Mirhosseini et al. [39] answered the question whether automated pull requests and badges that indicate out-of-date dependencies on platforms like *GitHub* can encourage developers to update dependencies. The authors found that badges provide a good incentive for developers to update, because there is a psychological urge to keep the badges green. Automated pull requests offer an easy and directly actionable way to update, but when there are many of them, notification fatigue can work against update discipline. In summary, badges account for a 1.4 times higher probability to update, while automated pull requests increase the probability by a factor of 1.6 according to [39]. A developer survey done by the authors also showed that a single bad update of one library can cause developers to be strongly reluctant from updating in the future. Therefore, releasing library versions comes with a high responsibility to not break software, in order to keep an environment where developers update their projects regularly to fix security vulnerabilities.

7. Discussion

This chapter discusses the results and findings of this thesis. First, the dangers and practical exploitability of real-world Go applications due to usages of the *unsafe* API are elaborated. Then, the patches that were sent to open-source code maintainers are described in detail, and suggestions towards a world with safer usage practices of *unsafe* code are given. Finally, improvements in upcoming versions of the Go programming language are presented, and threats to validity are discussed.

7.1. Practical Exploitability of *Unsafe* Code

This thesis took a deep look into the *unsafe* API of the Go programming language. It was shown that it is effectively used by a large fraction (90.96%) of the 343 analyzed top-starred open-source projects, either directly or by including *unsafe* usages through third-party dependencies. While there are many legitimate use cases for *unsafe* code, such as efficient conversions between otherwise incompatible types without allocating additional memory, or accessing the foreign function interface (FFI), a thorough manual study also showed that there are dangers and possible vulnerabilities that can come from it. The data collected when creating the labeled data set of annotated *unsafe* usage examples shows that the majority of the usages are safe and legitimate, but there were also several bugs that could lead to severe vulnerabilities.

A contribution of this thesis is novel evidence of possible misuses of the *unsafe* API with concrete proof showing how a vulnerability is introduced. The changes needed to avoid the misuses are often minor and subtle, which underlines the fact that *unsafe* is a fragile and dangerous feature of the Go programming language. It is complicated to understand the implications that a given piece of code has, and could have in the future, especially if there is interaction with other parts of the code base and other developers. If a bug is introduced, it can easily slip through code review and be part of a zero day exploit some time later. The bugs that were found during the course of this work, and to which patches were submitted as described in the next section, may seem few and minor at first glance, but it only takes one *use-after-free* bug at a critical position to open up the possibility of a major remote code execution attack.

In my opinion, the general direction that programming languages like Go or Rust take is the

right one. By enforcing a strict safety net in most cases and still allowing escape hatches to circumvent these measures when it is absolutely necessary, they combine the best of two worlds. In large parts of the code base, buffer overflows, race conditions etc. are mostly prevented at the small cost of a slightly more rigid coding workflow and developer training. Still, there is the option of complete flexibility and total control over the memory if it should be needed. This allows organizations to focus auditing and review efforts on the code parts that use the *unsafe* API, which is a fraction of the total code.

7.2. Patching Open-Source Projects

In the process of analyzing *unsafe* usages in real-world Go code, and using the novel static analysis tool *go-safer*, 64 unsafe-related bugs were found that can lead to security vulnerabilities. Most of them (63) are instances of incorrect constructions of slice header values, which are used with in-place conversions between slices of different types. These can cause *use-after-free* vulnerabilities due to the garbage collector freeing a value that is still in use, or an error in the escape analysis causing a value to be placed on the stack incorrectly, as described in Sections 3.2.2 and 3.2.3. Furthermore, there is the bug causing incorrect length information in a slice in the *go-fuse* library described in Section 3.3.2.

During the course of this work, 14 pull requests with patches to these bugs were submitted to the authors of the affected libraries on *GitHub*. These pull requests contain fixes to one or more bugs, adding up to a total number of 63 resolved bugs. Furthermore, one additional pull request with one more bug already existed. They are listed in Table 7.1 along with their respective *GitHub* projects. The Popularity column shows how many projects in the data set of popular Go projects use the respective library, which is related to the impact of the bugs. Furthermore, the table indicates which of the pull requests have been merged so far, and how many bugs they contain individually.

Sending pull requests on *GitHub* is a public disclosure procedure, although the bugs have not been announced on any news pages. Given that there are currently no actual exploits that use the bugs to inject code or leak data, this was a good choice compared to other procedures like a responsible disclosure, for example. Submitting *GitHub* pull requests allows the code authors to easily merge the necessary changes. So far, 10 of the pull requests have been reviewed, acknowledged, and accepted by the authors. Thus, 59 (92%) of the bugs have been fixed. The remaining were not rejected either, but received no attention due to a generally high volume of open pull requests in the project.

Table 7.1.: Pull requests on GitHub to fix vulnerable Go libraries

	Project	PR-Name	Popularity	Bugs	PR Number	Merged
1	hanwen/go-fuse	batch forget: fix missing return to handle malformed input data	3 projects	1	PR #363	no
2	buger/jsonparser	Fix possible memory confusion in unsafe slice cast	4 projects	1	PR #204	yes
3	elastic/go-structform	Fix possible memory…	1 project	2	PR #21	yes
4	go-fiber/utils	Fix possible memory…	1 project	1	PR #7	yes
5	influxdata/influxdb	fix: possible memory…	8 projects	1	PR #18307	no
6	influxdata/influxdb1-client	fix: possible memory…	4 projects	1	PR #40	no
7	modern-go/reflect2	Fix possible memory…	71 projects	1	PR #13	yes
8	savsgio/gotils	Fix possible memory…	1 project	2	PR #2	yes
9	valyala/fasttemplate	Fix possible memory…	10 projects	1	PR #21	yes
10	weaveworks/ps	Fix possible memory…	1 project	1	PR #3	no
11	yuin/goldmark	Fix possible memory…	5 projects	1	PR #134	yes
12	yuin/gopher-lua	Fix possible memory…	6 projects	1	PR #287	yes
13	gorgonia/tensor	Fix possible memory…	1 project	1	PR #68	yes
14	gorgonia/tensor	Fix 48 more possible…	1 project	48	PR #72	yes
15	mailru/easyjson	fix unsafe unsafe usage *(previously existing)*	42 projects	1	PR #258	no
	Total			**64**		**10 / 15**

7.3. Suggestions for Safer Go Code

Usages of *unsafe* that get imported through third-party dependencies are dangerous, because they can be hidden at first glance, but still get compiled into the resulting application binary. On top of the fact that external dependencies tend to get updated rather late, if at all, as was discussed in Section 6.5, there is a possibility that there is simply nobody even aware of the potential danger. The novel tool *go-geiger* is a step in the journey of improving this situation, as it allows to quickly and effectively differentiate the libraries that need auditing from those that do not. An effective next step would be to use this information even before deciding which library to use. If there are multiple external libraries that achieve the same goal, choosing which one to use should incorporate the data about how many *unsafe* usages each library contains, if any. Libraries with less *unsafe* code should be preferred over those with a lot. Previous work has proposed various code metrics to help choosing a specific library [12]. They include security and fault-proneness, both of which could be influenced by the use of *unsafe*. It would be beneficial to contribute towards a situation where library maintainers use the number of *unsafe* usages in their library as a feature or part of its advertisement. Similar to code quality report badges on the *GitHub* repository of the project, there could be a badge showing if the library uses any *unsafe* code, and if it does how much. However, it is important to remember that the number of *unsafe* usages is not itself an indicator for the safety of a library or project. First, security problems can also be introduced in code that does not use *unsafe* at all. Second, there could be

many well-audited *unsafe* usages in one project, and a single one that causes a vulnerability in another one. Although the number of *unsafe* usages in the second project is smaller, it might be much less secure.

7.4. Improvements in Upcoming Go Releases

Since the study data for this thesis has been collected, the most recent Go release 1.15 from August 2020 has enabled additional static checking of *unsafe.Pointer* usages with the *-d=checkptr* compiler flag[1]. The development of the new check had been started with Go release 1.14 already. It checks that the alignment of the target type matches the alignment of the pointer when converting *unsafe* pointers to an arbitrary type. Furthermore, the result of pointer arithmetic can not be a completely arbitrary heap value anymore, instead at least one argument of the arithmetic expression must already be a pointer into the resulting value. Both checks prevent some misuses of *unsafe* that can occur if the developer took a false assumption over the types involved in a conversion, and therefore they are a valuable step towards a safer usage of *unsafe* in Go. Architecture-dependent types as covered in Section 3.1 are less dangerous with this new check, because it detects a mismatch in memory alignment or byte order when the application is compiled for an unsuitable architecture. However, the remaining exploits discussed in Chapter 3 remain possible with the new check. Also, it can not find the misuses that are detected by *go-safer*, which underlines the need of continuous improvement of the developer tools and continued awareness of potentially still unknown dangers with the use of *unsafe*.

There is an open proposal for introducing a new type *unsafe.Slice* into the Go *unsafe* package[2]. It is supposed to replace the *reflect.SliceHeader* type and differs in the type of the *Data* field. While that field is a *uintptr* in the *reflect* package, the proposed type uses *unsafe.Pointer*. Thus, the new representation for a slice stores the reference to the underlying array in a type that is treated as a pointer by the garbage collector and escape analysis. This solves the slice vulnerabilities described in Sections 3.2.2 and 3.2.3, which are rather common in real Go applications. Therefore, it would provide a significantly safer way of writing such code. By the time this thesis is submitted, the proposal is actively discussed and patches to the Go compiler have already been accepted. The feature is currently planned to be included in the 1.17 release of Go, which would be expected to be finished around August, 2021.

Finally, some *unsafe* usages in the analyzed open-source projects were necessary to achieve functionality that could have been implemented with generics, if they were available in Go. Generics are in fact one of the most widely requested features for the Go programming language, with 79% of participants in the 2019 Go Developer Survey[3], who answered the particular

[1] `https://golang.org/doc/go1.14#compiler`
[2] `https://github.com/golang/go/issues/19367`
[3] `https://blog.golang.org/survey2019-results`

question, stating that they think it is a critical missing feature. Since generics have now been officially announced[4] for the upcoming Go release 2.0, these *unsafe* usages can then be replaced by the new generics support. Similarly, other language improvements such as a new foreign function interface could make other usages of *unsafe*, which are necessary at the moment, obsolete, decreasing the room for errors made by the developers and increasing safety.

7.5. Threats to Validity

This section discusses potential threats to the validity of this work. Internal threats concern the quality of execution, or whether any results could be explained by other factors that were not covered in this thesis. The empirical study and quantitative evaluation presented in Section 4.3 focused on code for the *amd64* architecture. This is due to a limitation of *go-geiger*, which currently only finds *unsafe* usages in code targeted for the architecture that *go-geiger* is executed on. There could possibly be a different prevalence of *unsafe*, or a contradictory distribution of *unsafe* token types, on other architectures. This threat is mitigated first by the fact that *amd64* is by far the most commonly used architecture in the data set presented in this thesis, therefore code for other architectures should not have a large influence on the statistics. Secondly, an analysis using *grep* showed similar results to the study using *go-geiger*. While *grep* can not exclude unreachable code, it includes code for all architectures.

Furthermore, the manual analysis of *unsafe* code patterns could have missed some vulnerabilities. In fact, it is almost guaranteed that there are more ways to exploit *unsafe* usages. However, the objective of this thesis is not a necessarily exhaustive search for all possible exploit vectors. Instead, the contributions of this work include an analysis that did reveal multiple vulnerabilities, thus providing a step towards safer Go applications.

On the other hand, external threats to validity concern the ability to generalize the study results to new data. First, the raw data for the empirical study in Section 4.3, as well as the labeled data set described in Section 4.4, was gathered from the most popular projects on *GitHub*, measured by their number of stars. Furthermore, projects without support for the Go module system and projects that would not compile on our machines were excluded. It is possible that the usage of *unsafe* is different in less popular projects. This is especially true for projects with fewer stars, where there might not be as many developers reviewing *unsafe* usages due to a smaller public interest. Projects which have not yet adopted the module system or contain build errors could have a generally lower code quality, which means there could be a bias in the study data towards higher-quality projects. However, choosing projects by popularity mitigates possible bias towards specific domains of projects, such as virtualization or databases, because the number of stars is an easy and rather neutral metric. Therefore, the project set enables a good overview on the usage of *unsafe* in a broad selection of projects.

[4] `https://blog.golang.org/go2-next-steps`

8. Conclusion

In this thesis, a detailed study of the *unsafe* API provided by Google's Go programming language with respect to a security context was conducted. A thorough analysis of the dangers that come with its usage was presented, including proof-of-concept exploits that show how possible vulnerabilities can be exploited in practice. This analysis particularly looked at three main areas of dangers. Types with platform-dependent types such as *int* can cause problems when they are converted to other types directly, because the underlying memory might not align when the project is compiled for a different architecture, or the memory layout of the types after compilation is changed in future versions of Go. Incorrect conversions of slices through the internal slice representation used by the Go runtime can cause the garbage collector to miss a live value, or the Go compiler's escape analysis algorithm to misplace a value on the stack rather than the heap, both of which can cause non-deterministic or deterministic use-after-free vulnerabilities, respectively. Buffer overflow vulnerabilities, which can be introduced by constructing slice values from unsuitable buffers, can lead to severe consequences such as information leak or code injection vulnerabilities.

Thus, it is needed to audit usages of the *unsafe* API in projects. Such usages can be introduced through dependencies, in fact most are. Therefore, dependencies must be checked, too. The novel static code analysis tool *go-geiger*, which can help developers with this task, was presented. It can identify and count *unsafe* usages in Go packages, including their dependencies. This allows developers to focus their *unsafe* audit efforts to the most sensible packages. Such a tool was previously available for other languages, but not for Go. Using *go-geiger*, an empirical study on the current state of *unsafe* usage in 343 of the top 500 most-starred open-source Go projects on *GitHub* was presented. This study revealed that 38.19% of the projects contain *unsafe* usages, however 90.96% include *unsafe* code through third-party dependencies. Of the 62,025 packages that were analyzed in total, 5.5% contained *unsafe* code. The average depth in the dependency tree of 3.08 showed that most packages with *unsafe* usages are reasonably close to the root module. However, it is still hard to manually audit the complete code base including external libraries, highlighting the importance of support by developer tools for this task.

A novel data set of 1,400 manually labeled *unsafe* code samples was created. They are classified in two dimensions, on what is being done and for what purpose. This data set shows that the the most common reasons for using *unsafe* are optimizations and efficiency, interoperability with

external libraries, or to circumvent language limitations.

Furthermore, *go-safer* was presented, a novel *go vet*-style linter that is focused around *unsafe* code. It can identify two dangerous and common usage patterns: incorrect conversions of slice and string header values to actual slices or strings, which can introduce use-after-free vulnerabilities, and in-place conversions of types that have platform-dependent sizes or memory layouts. The performance of *go-safer* was evaluated both on the new labeled data set of *unsafe* usages and on a set of open-source Go packages, selected by their number of *unsafe* usages and lines of code, which were reviewed manually. In this evaluation, *go-safer* achieved excellent results, with an accuracy of 95.5% on the labeled data set and 99% on the set of packages. Its high precision allows it to be used productively. Thus, *go-safer* is a valuable tool that can be added next to existing tools like *go vet* and *gosec*, which are not good at detecting the *unsafe* misuses *go-safer* finds, with a large number of false negatives and false positives, respectively. Using *go-safer*, 64 bugs in open-source Go libraries were found. To fix them, 14 pull requests have been submitted to the authors, and one already existed. So far, 10 pull requests have been merged, fixing 59 (92%) of the bugs.

In summary, *unsafe* code is commonly used in the most popular Go projects for a number of reasons such as efficiency or the foreign function interface. Using the novel static analysis tools contributed in this thesis, developers can embrace this fact and mitigate the risks that come with it by effectively localizing and checking *unsafe* usages in their own and third-party code. Additionally, the novel evidence of how to actually exploit possible vulnerabilities related to *unsafe* code helps developers to understand the dangers and, ultimately, avoid them.

9. Future Work

Based on the contributions of this thesis, future work might look into the following areas. The novel data set of manually labeled *unsafe* usages can be used for interesting further research. First, the code samples in the *efficiency* class do not strictly require the use of *unsafe*, but should improve the performance of the code. Future work could quantify by how much *unsafe* actually increases the efficiency compared to using other, memory-safe language features of Go. If the benefit is small, *unsafe* would pose an unnecessary risk that could be avoided. Especially, a comparison with performance improvements of new Go compiler releases would be interesting to see if simply waiting for a new version of the compiler can provide more efficient program binaries than *unsafe* does. Furthermore, future work could evaluate the possibility of automatically replacing such *unsafe* usages with safe language constructs. The labeled data set could also be used as a training and verification data set in machine learning applications. For example, an automated classification tool that predicts the purpose and danger of unseen *unsafe* code patterns could be built. This would be valuable both for security analysts to conduct a large-scale study of *unsafe* usages in thousands of projects, and for software engineers to highlight particularly dangerous usages in the findings of *go-geiger*, as well as ones that can be replaced with new language features.

Next, future research could focus on verification techniques for *unsafe* Go code, such as annotations for contracts or invariants that formalize assertions about control and data flow. For example, *unsafe* code that creates implicitly read-only values could be formally declared immutable, so that a verifier can detect invalid accesses to the value. A static verification such as the Viper framework [40] would probably be preferable over a dynamic one, because the additional runtime overhead could potentially outweigh the efficiency improvements of using *unsafe* in the first place.

Finally, establishing a public documentation of good and bad usage examples of the Go *unsafe* API could help developers avoid common mistakes. For insecure usages, it could provide an explanation of the potential vulnerability, as well as a suggestion for a secure alternative. Similar to *go-safer*, a linter could be developed to identify harmless usages of *unsafe*, which could automatically be flagged safe and thus be excluded from manual review.

A. Appendix

Table A.1.: Open-source Go projects on GitHub used for the empirical study on unsafe

(blank): included for analysis, *: used for labeled data set, nm: no module support, bf: build failed

	Name	Stars	Forks	Revision	
1	golang/go	72,988	10,460	6bf2eea62a	*nm*
2	kubernetes/kubernetes	66,512	23,806	fb9e1946b0	*
3	moby/moby	57,189	16,540	763f9e799b	*nm*
4	avelino/awesome-go	54,733	7,267	3e27d63fe2	*nm*
5	gohugoio/hugo	44,317	5,049	6a3e89743c	
6	gin-gonic/gin	38,459	4,441	5e40c1d49c	
7	fatedier/frp	36,184	6,860	2406ecdfea	
8	astaxie/build-web-applica…	34,787	9,516	606abd586a	*nm*
9	gogs/gogs	34,522	4,030	7e99a6ce42	
10	v2ray/v2ray-core	31,853	7,270	edb4fed387	
11	syncthing/syncthing	31,357	2,686	04ff890263	
12	etcd-io/etcd	31,265	6,523	9b6c3e3378	
13	prometheus/prometheus	31,006	4,725	83619aa9ac	
14	junegunn/fzf	29,549	1,207	f81feb1e69	
15	containous/traefik	28,977	3,178	7928e6d0cd	
16	caddyserver/caddy	28,739	2,334	9415feca7c	
17	ethereum/go-ethereum	26,010	9,492	389da6aa48	
18	FiloSottile/mkcert	24,272	1,003	a2b1208e9c	
19	astaxie/beego	23,997	4,825	8f3d1c5f42	
20	iikira/BaiduPCS-Go	23,924	3,974	a829eef6b0	
21	pingcap/tidb	23,680	3,572	777907bcee	
22	istio/istio	22,946	4,312	13855a94f9	
23	hashicorp/terraform	22,151	5,729	01f91316da	
24	minio/minio	22,065	2,263	231c5cf6de	
25	rclone/rclone	21,733	1,733	764b90a519	
26	unknwon/the-way-to-go_ZH_…	21,596	5,939	eec6f83fcb	*nm*
27	drone/drone	21,111	2,058	aa84e311c0	
28	wagoodman/dive	20,197	727	103b05f3c0	
29	go-gitea/gitea	19,872	2,342	9f55769804	
30	github/hub	19,698	2,039	93537d4575	
31	hashicorp/consul	19,249	3,308	6c444ba24c	
32	influxdata/influxdb	18,970	2,680	41156ca646	
33	inconshreveable/ngrok	18,634	3,287	a8e7fa4863	*nm*
34	jinzhu/gorm	18,545	2,144	7ea143b548	
35	kubernetes/minikube	18,317	2,982	ea20609a3a	
36	mattermost/mattermost-ser…	18,277	4,157	e83cc7357c	*
37	cockroachdb/cockroach	18,260	2,177	e148388d9e	*nm*
38	kataras/iris	18,217	2,028	b6ac39480b	
39	nsqio/nsq	17,777	2,350	8db8c50e5e	
40	openfaas/faas	17,681	1,463	6841fba36b	*nm*
41	labstack/echo	17,310	1,561	43e32ba83d	
42	helm/helm	17,197	5,126	15d9a6190f	
43	spf13/cobra	17,160	1,472	94a87a7b83	
44	go-kit/kit	17,032	1,783	81a2d1f550	
45	yeasy/docker_practice	16,880	4,642	ddec6641b9	*nm*
46	jesseduffield/lazygit	16,418	581	cf5cefb2d6	
47	hashicorp/vault	15,810	2,447	2b8cac7260	
48	jesseduffield/lazydocker	15,109	565	10617da560	
49	sirupsen/logrus	14,940	1,674	6699a89a23	
50	joewalnes/websocketd	14,653	829	2190c8ab4c	
51	tsenart/vegeta	14,648	938	d9b795aec8	
52	rancher/rancher	14,344	1,758	56a464049e	*
53	go-delve/delve	13,968	1,363	4a9b3419d1	
54	yudai/gotty	13,964	1,032	a080c85cbc	*nm*
55	urfave/cli	13,795	1,150	477292c8d4	
56	zyedidia/micro	13,488	657	8956448fca	
57	cayleygraph/cayley	13,411	1,208	ab2941b35e	
58	helm/charts	13,315	14,774	f30b0d5c24	*nm*
59	dgraph-io/dgraph	13,262	946	4f0645ec7d	

Continued

Continued

	Name	Stars	Forks	Revision	
60	golang/dep	13,213	1,104	87f309484f	*nm*
61	micro/go-micro	13,093	1,368	3f354f3c30	
62	rancher/k3s	12,868	973	b237637338	*bf*
63	buger/goreplay	12,826	1,244	5c5ef3ac15	
64	tmrts/go-patterns	12,758	1,193	f978e42036	*nm*
65	chai2010/advanced-go-prog…	12,696	2,105	00bc3e0868	*bf*
66	coreybutler/nvm-windows	12,692	1,252	4ea3bc0c13	*nm*
67	valyala/fasthttp	12,530	1,041	2f92c68a07	
68	spf13/viper	12,351	1,130	13df721090	
69	ehang-io/nps	12,243	2,112	f391813a28	*bf*
70	gorilla/websocket	11,989	2,047	b65e62901f	
71	gorilla/mux	11,982	1,208	948bec34b5	
72	goharbor/harbor	11,944	3,194	7c2bfb1378	*nm*
73	xtaci/kcptun	11,849	2,308	912a97993e	
74	revel/revel	11,708	1,359	a3d7a7c23c	*nm*
75	txthinking/brook	11,463	2,160	e8f20902a9	*nm*
76	wtfutil/wtf	11,461	632	204e286caa	
77	kubernetes/kops	11,459	3,583	4b4dbd4285	
78	grpc/grpc-go	11,361	2,400	e0ec2b8320	
79	julienschmidt/httprouter	11,269	1,078	8c9f31f047	
80	CodisLabs/codis	11,106	2,497	de1ad026e3	*nm*
81	quii/learn-go-with-tests	11,075	1,331	3e2a9c5a98	
82	jaegertracing/jaeger	10,981	1,196	06c8f7cc82	
83	gocolly/colly	10,956	925	556442d455	
84	go-martini/martini	10,953	1,104	22fa46961a	*nm*
85	fogleman/primitive	10,825	529	0373c21645	*nm*
86	google/cadvisor	10,743	1,578	8d4d5ea98d	
87	boltdb/bolt	10,715	1,253	fd01fc79c5	*nm*
88	stretchr/testify	10,531	901	004e3cb722	
89	peterq/pan-light	10,492	2,342	867eee7a92	
90	iawia002/annie	10,444	1,141	dcf8152855	
91	hyperledger/fabric	10,248	5,897	41f8b0a033	
92	restic/restic	10,201	712	cba6ad8d8e	
93	hashicorp/packer	10,182	2,807	193395d734	
94	vitessio/vitess	10,080	1,305	734ed78e52	
95	google/grumpy	10,060	650	3ec8795918	*nm*
96	google/gvisor	9,962	714	a8c1b32660	*bf*
97	bcicen/ctop	9,945	381	4741b276e4	
98	gizak/termui	9,864	613	4cca61d83f	*nm*
99	go-kratos/kratos	9,844	2,106	521d240568	
100	uber-go/zap	9,826	736	e931a6bb13	
101	hoanhan101/ultimate-go	9,753	736	e91ebe79b9	
102	ipfs/go-ipfs	9,721	1,784	285d743173	
103	fyne-io/fyne	9,719	445	e7f1331819	
104	GoogleContainerTools/skaf…	9,712	945	dc44de4d42	
105	chrislusf/seaweedfs	9,639	1,277	6286a454c7	
106	dutchcoders/transfer.sh	9,576	984	92055f1b3c	
107	grafana/loki	9,537	922	10a1f28a85	
108	go-sql-driver/mysql	9,522	1,692	8c3a2d9049	
109	gopherjs/gopherjs	9,393	445	fce0ec30dd	*nm*
110	esimov/caire	9,286	345	8555afc75e	
111	cli/cli	9,025	365	675cbbc2a4	
112	asciimoo/wuzz	8,981	341	f087795f72	*nm*
113	evanw/esbuild	8,946	168	9863668f2f	
114	rkt/rkt	8,883	863	171c416fac	*nm*
115	Dreamacro/clash	8,880	1,260	3638b077cd	
116	go-redis/redis	8,861	1,142	a999d1ecd8	
117	elastic/beats	8,852	3,207	df6f2169c5	*
118	snail007/goproxy	8,842	1,792	029cc0d002	*nm*
119	PuerkitoBio/goquery	8,836	698	89946c829f	
120	golang/groupcache	8,725	1,013	8c9f03a8e5	*nm*
121	influxdata/telegraf	8,644	3,493	430854f6de	
122	ory/hydra	8,643	786	70ab4fff18	
123	Netflix/chaosmonkey	8,626	629	68e3282ef7	*nm*
124	docker-slim/docker-slim	8,620	294	79490f5f1c	
125	grpc-ecosystem/grpc-gatew…	8,523	1,119	0a916456be	
126	rakyll/hey	8,414	634	f3676ef133	
127	jmoiron/sqlx	8,394	658	ee514944af	
128	dinedal/textql	8,359	285	1d6fef53e0	
129	emirpasic/gods	8,352	955	80e934ed68	*nm*
130	git-lfs/git-lfs	8,213	1,560	7dfe9fe110	

Continued

Continued

	Name	Stars	Forks	Revision	
131	gravitational/teleport	8,112	622	4bc0346518	*nm*
132	cyfdecyf/cow	8,074	1,644	41c0fb157c	*nm*
133	micro/micro	8,041	683	d607618d5b	
134	apex/apex	8,024	588	16c08f0eb3	
135	Masterminds/glide	8,004	538	b94b39d657	*nm*
136	sqshq/sampler	8,003	385	1af22f6bfd	
137	kubernetes/dashboard	7,882	2,214	cd7a6156b1	
138	nats-io/nats-server	7,757	762	f859edaf4f	
139	apex/up	7,753	314	5777a4ff5f	
140	github/gh-ost	7,734	768	4dab06e92b	*nm*
141	json-iterator/go	7,702	627	55287ed53a	
142	dgraph-io/badger	7,702	615	00b86b2e74	
143	dgrijalva/jwt-go	7,697	699	dc14462fd5	*nm*
144	kubernetes/ingress-nginx	7,693	3,667	f6f695e7a1	
145	flynn/flynn	7,625	567	755c95684f	
146	go-chi/chi	7,525	503	5704d7ee98	*nm*
147	future-architect/vuls	7,505	827	835dc08049	
148	andlabs/ui	7,487	674	867a9e5a49	*nm*
149	openshift/origin	7,466	4,288	1bc7b9e328	*nm*
150	gomodule/redigo	7,428	1,065	2eadaa0e59	
151	ahmetb/kubectx	7,413	501	401188fefd	
152	therecipe/qt	7,393	543	5074eb6d8c	*bf*
153	talk-go/night	7,386	711	cdd02dcd04	
154	ardanlabs/gotraining	7,350	1,503	86a37cbcfd	
155	bettercap/bettercap	7,293	745	6725a2aa53	
156	unknwon/go-fundamental-pr…	7,272	1,884	5542bcd8fd	*nm*
157	rook/rook	7,208	1,472	ff90fa7098	*
158	sjwhitworth/golearn	7,199	1,009	3e43e74895	*nm*
159	attic-labs/noms	7,194	266	39057233bf	
160	google/go-cloud	7,052	544	bee021b0e3	
161	claudiodangelis/qrcp	7,018	411	c6b4c7454b	
162	kelseyhightower/confd	6,992	1,178	cccd334562	*nm*
163	opencontainers/runc	6,929	1,297	4f0bdafc8a	
164	tidwall/tile38	6,883	413	9c6be0f78e	
165	keybase/client	6,870	956	9e24cdae97	*nm*
166	cjbassi/gotop	6,819	272	61ed1ad0c3	
167	casbin/casbin	6,819	755	3df797354a	
168	derailed/k9s	6,805	365	68da73952e	
169	filebrowser/filebrowser	6,794	1,058	6e5405eeed	
170	polaris1119/The-Golang-St…	6,783	1,644	802818f4f3	*nm*
171	urfave/negroni	6,666	540	e1cd4e5018	
172	golang/protobuf	6,663	1,248	d04d7b157b	
173	shadowsocks/shadowsocks-g…	6,662	3,577	3e585ff906	*nm*
174	tylertreat/comcast	6,649	296	36c754a9a7	*nm*
175	quay/clair	6,650	840	db2e7d5c0c	
176	blevesearch/bleve	6,586	506	2b80a2aedf	
177	gdrive-org/gdrive	6,582	834	8e12e1c1c9	*nm*
178	loadimpact/k6	6,567	372	479707f90e	*nm*
179	usefathom/fathom	6,479	287	69baac5c4a	*nm*
180	hybridgroup/gobot	6,469	815	9d38858276	
181	mailhog/MailHog	6,455	457	d70a557574	*nm*
182	dapr/dapr	6,447	380	592af5afb9	
183	tidwall/gjson	6,424	420	f042915ca1	
184	henrylee2cn/pholcus	6,423	1,581	bf4a87b9aa	*bf*
185	graphql-go/graphql	6,419	551	116f19d099	
186	sosedoff/pgweb	6,398	468	111b1720b4	*nm*
187	tomnomnom/gron	6,383	137	602235e754	
188	fabiolb/fabio	6,355	555	c034d4fae1	
189	jroimartin/gocui	6,324	404	c055c87ae8	*nm*
190	tinygo-org/tinygo	6,305	298	a9ba6ebad9	*bf*
191	360EntSecGroup-Skylar/exc…	6,301	685	2ae631376b	
192	inlets/inlets	6,295	351	ae73b7d20d	*nm*
193	schachmat/wego	6,291	396	994e4f1417	*nm*
194	linuxkit/linuxkit	6,276	791	9d5a22d44a	*nm*
195	direnv/direnv	6,249	324	1cf67502b5	
196	crawlab-team/crawlab	6,245	853	d75a993121	*nm*
197	hashicorp/nomad	6,207	1,078	5be192fac3	*nm*
198	XiaoMi/soar	6,138	894	078704875f	*nm*
199	coredns/coredns	6,075	1,017	bc2ba28865	
200	go-xorm/xorm	6,044	763	f39e5d9bfd	
201	robfig/cron	6,029	975	6a8421bcff	
202	DNSCrypt/dnscrypt-proxy	6,017	579	3d5f877058	
203	cheat/cheat	5,970	619	8e602b0e93	
204	GoogleContainerTools/kani…	5,962	582	e0f93578b6	
205	lib/pq	5,922	735	d408f9c948	
206	GoesToEleven/GolangTraini…	5,918	2,303	afa19f5c43	*nm*
207	docker/machine	5,895	1,708	b170508bf4	*nm*
208	peco/peco	5,891	188	9c4464680c	
209	rancher/os	5,885	606	2ec7f4fc9e	*nm*
210	rqlite/rqlite	5,884	322	5f762f01ec	
211	open-falcon/falcon-plus	5,874	1,332	850e7f79d4	*nm*
212	xo/usql	5,871	195	bdff722f7b	*
213	zricethezav/gitleaks	5,871	470	e0f6399d5c	
214	weaveworks/weave	5,840	577	0d4099bba5	*nm*
215	docker/classicswarm	5,837	1,137	613bdb260e	*nm*
216	google/go-github	5,832	1,283	a86f060e61	
217	containrrr/watchtower	5,832	365	cd21516709	

Continued

Continued

	Name	Stars	Forks	Revision	
218	teh-cmc/go-internals	5,831	253	dd22691012	*nm*
219	aws/aws-sdk-go	5,811	1,476	645efefb5b	
220	Shopify/sarama	5,806	1,047	b5764af1c4	
221	thanos-io/thanos	5,793	777	63ef382fc3	
222	pkg/errors	5,792	419	614d223910	*nm*
223	kubeless/kubeless	5,742	586	78d3cedd20	
224	pulumi/pulumi	5,727	299	45e1917a30	*nm*
225	ginuerzh/gost	5,726	1,101	2707a8f0a9	
226	jetstack/cert-manager	5,671	1,001	78ee463a98	
227	containerd/containerd	5,656	1,169	8e9ba8376e	*nm*
228	tools/godep	5,640	478	ce0bfadeb5	*nm*
229	smartystreets/goconvey	5,627	425	505e419363	
230	goreleaser/goreleaser	5,622	398	c31cc85cb2	
231	gobuffalo/buffalo	5,598	440	b30b19999a	
232	Workiva/go-datastructures…	5,598	654	0819bcaf26	*nm*
233	argoproj/argo	5,583	909	09092147cf	
234	linkerd/linkerd2	5,574	541	9a02e0d300	
235	gofiber/fiber	5,566	199	38c8bb240d	
236	kubernetes-sigs/kind	5,557	619	94031cdb85	
237	simeji/jid	5,554	114	be32ff3178	
238	nektos/act	5,552	136	39667011b8	
239	cdr/sshcode	5,516	213	50e859cd10	
240	upspin/upspin	5,514	271	5ddde7b8e6	
241	cilium/cilium	5,501	626	9b0ae85b5f	*
242	slackhq/nebula	5,487	308	ff13aba8fc	
243	TykTechnologies/tyk	5,475	719	ce0ee257b6	
244	kubernetes-sigs/kustomize…	5,475	985	c1a2bf14da	*nm*
245	erroneousboat/slack-term	5,446	183	c19c20a3e5	
246	coreos/flannel	5,406	1,665	485649719b	*nm*
247	zserge/lorca	5,387	236	a3e43396a4	
248	photoprism/photoprism	5,386	320	a77b2431d3	
249	go-playground/validator	5,355	478	ea924ce89a	
250	golangci/golangci-lint	5,347	483	71b2f04e88	
251	robertkrimen/otto	5,343	469	c382bd3c16	*nm*
252	filhodanuvem/gitql	5,330	140	875d794ca2	
253	docker/distribution	5,326	1,758	742aab907b	
254	yeasy/blockchain_guide	5,311	1,716	c4d54e552c	*nm*
255	adnanh/webhook	5,281	437	345bf3d409	
256	cloudflare/cfssl	5,272	705	6b49beae21	
257	gcla/termshark	5,229	231	32a152f537	
258	go-ego/riot	5,228	367	a6e6be1c36	
259	kubernetes/kompose	5,211	438	6b018fab7b	*nm*
260	flike/kingshard	5,187	1,039	e5cad209c4	
261	go-swagger/go-swagger	5,181	817	c1166eb277	
262	fission/fission	5,136	477	9dd084810f	
263	src-d/go-git	5,052	563	8b0c2116ce	
264	olivere/elastic	5,031	902	639b7f6e0c	
265	google/seesaw	5,027	471	2a7e69077c	
266	kardianos/govendor	5,019	402	e31350db97	*nm*
267	tektoncd/pipeline	5,016	823	cc724c8a79	
268	yinghuocho/firefly-proxy	4,988	933	69c2bd0414	*nm*
269	akavel/up	4,968	100	5786314a2b	
270	dragonflyoss/Dragonfly	4,961	656	e8a651459f	
271	bitly/oauth2_proxy	4,953	1,173	fa2771998a	*nm*
272	concourse/concourse	4,934	606	82a76b6492	
273	shirou/gopsutil	4,931	880	42aec722ba	*nm*
274	hashicorp/serf	4,921	504	5642cc7572	
275	ponzu-cms/ponzu	4,906	327	9bc41b7031	*nm*
276	nsf/gocode	4,865	665	5bee97b488	*nm*
277	cortexlabs/cortex	4,859	380	aae3ff7800	
278	perkeep/perkeep	4,828	365	c2e31370dd	
279	rakyll/boom	4,814	347	e99ce27f08	*nm*
280	fluxcd/flux	4,812	850	3b40b58dfd	
281	Jguer/yay	4,809	180	db3993b83b	
282	cloudreve/Cloudreve	4,774	925	14f5982b47	
283	chromedp/chromedp	4,765	414	3976e2ae9c	
284	schollz/find	4,747	357	b69e6fd241	*nm*
285	gotify/server	4,741	225	93b30c5c44	
286	OpenDiablo2/OpenDiablo2	4,721	314	515b66736d	
287	davecheney/httpstat	4,698	245	474bdf475e	
288	gophish/gophish	4,684	840	ec8b17238e	
289	fiorix/freegeoip	4,665	764	4693d60f74	*nm*
290	inancgumus/learngo	4,661	535	65e854c4ae	
291	appleboy/gorush	4,642	512	6bad4797b4	
292	smallnest/rpcx	4,640	766	57cd85a346	
293	prometheus/node_exporter	4,603	1,158	b9c96706a7	
294	wallix/awless	4,600	232	44e892b496	*nm*
295	miekg/dns	4,597	748	203ad2480b	
296	fnproject/fn	4,595	345	d55e01ab7d	
297	pion/webrtc	4,587	510	4618922011	
298	facebookarchive/grace	4,561	362	75cf193824	*nm*
299	google/battery-historian	4,556	823	d2356ba4fd	*nm*
300	containers/libpod	4,549	539	e8818ced80	*
301	Shopify/toxiproxy	4,547	252	10f0561d74	
302	michenriksen/gitrob	4,543	658	7be4c5306a	*nm*
303	dexidp/dex	4,536	964	0a9f56527e	
304	lxn/walk	4,523	668	55ccb3a9f5	*nm*

Continued

Continued

	Name	Stars	Forks	Revision	
305	99designs/gqlgen	4,479	441	40570d1b4d	
306	lightningnetwork/lnd	4,468	1,263	703e8acb36	
307	fogleman/nes	4,465	393	c94772f158	
308	fsnotify/fsnotify	4,459	518	7f4cf4dd2b	
309	MichaelMure/git-bug	4,456	144	5029cc1e76	
310	muesli/beehive	4,435	220	86804e3b1e	
311	mongodb/mongo-go-driver	4,434	498	0c7911fa65	
312	qor/qor	4,416	634	457d2e3f50	*nm*
313	pachyderm/pachyderm	4,410	417	2969188a6c	
314	golang-migrate/migrate	4,407	437	7236e82911	
315	google/gxui	4,403	303	f85e0a97b3	*nm*
316	jpillora/cloud-torrent	4,373	1,453	1a741e3d8d	*nm*
317	russross/blackfriday	4,370	533	abb995c466	
318	weaveworks/scope	4,354	554	bf90d56f0c	*
319	mozilla/sops	4,343	292	4bc27f6eb7	
320	gliderlabs/registrator	4,335	854	4322fe0030	*nm*
321	rivo/tview	4,324	258	fe95322038	
322	centrifugal/centrifugo	4,312	376	3cffbd1c8e	
323	docker/docker-ce	4,309	1,129	f3673004e3	*nm*
324	coreos/prometheus-operato…	4,303	1,949	b0ab8afe43	
325	uber/prototool	4,266	242	a6d064684c	
326	spiral/roadrunner	4,254	223	af2b441928	
327	terraform-providers/terra…	4,251	3,966	64fd8eb362	
328	coyove/goflyway	4,245	692	dfffaed0d1	*nm*
329	adonovan/gopl.io	4,237	1,666	65c318dde9	*nm*
330	alibaba/pouch	4,227	945	6ef73ce06b	*nm*
331	eranyanay/1m-go-websocket…	4,215	389	6758771e40	
332	tealeg/xlsx	4,213	676	d2b2b0d0a4	
333	Terry-Mao/goim	4,209	1,181	a8942503cb	
334	asaskevich/govalidator	4,197	416	21a406dcc5	
335	google/git-appraise	4,180	131	e9b94fd0f2	*nm*
336	tidwall/evio	4,176	328	399d938449	*nm*
337	golang/mock	4,176	344	92f53b0a56	
338	geektutu/7days-golang	4,155	458	689a6d01b7	*nm*
339	golang/mobile	4,143	514	4c31acba00	
340	MontFerret/ferret	4,135	209	94f13c66af	
341	mmcgrana/gobyexample	4,122	817	448c597a58	*nm*
342	RichardKnop/machinery	4,109	528	6d3fd3e756	
343	AdguardTeam/AdGuardHome	4,095	461	32d1f385ff	
344	lucas-clemente/quic-go	4,079	476	85c19fbb5a	
345	gaia-pipeline/gaia	4,065	189	2654f288c7	
346	golang/tools	4,063	1,430	6be401e3f7	
347	twitchtv/twirp	4,058	196	20edcc73bb	*nm*
348	google/gops	4,053	234	ccfbf8d4a	
349	go-acme/lego	4,043	539	7c3689d08a	
350	lifei6671/mindoc	4,037	1,159	b7d8de8e93	*nm*
351	go-yaml/yaml	4,031	678	0b1645d91e	
352	gliderlabs/logspout	4,020	646	301ffe1253	
353	vmware-tanzu/octant	3,979	265	ec4170daf8	
354	EasyDarwin/EasyDarwin	3,972	1,668	8637f73e52	*nm*
355	jpillora/chisel	3,968	468	5271d7b062	
356	go-flutter-desktop/go-flu…	3,964	202	cc310c12f7	
357	goproxyio/goproxy	3,962	252	ead1140c92	
358	aquasecurity/trivy	3,933	337	78b7529172	
359	shazow/ssh-chat	3,925	322	3c4d90cfc1	
360	hashicorp/consul-template…	3,914	579	b7641ebcd3	
361	gruntwork-io/terratest	3,913	521	16f2895ca3	
362	microsoft/ethr	3,912	237	c1ace168b8	
363	onsi/ginkgo	3,903	376	9304a8cd49	
364	vmware-tanzu/velero	3,903	614	e7b668af2a	
365	Azure/draft	3,900	414	3206e97af5	*nm*
366	aelsabbahy/goss	3,898	335	06495afc9d	
367	patrickmn/go-cache	3,889	509	46f4078530	*nm*
368	allegro/bigcache	3,877	319	ccdbc60304	
369	dropbox/godropbox	3,871	395	52ad444d35	
370	gruntwork-io/terragrunt	3,868	504	c7694a5534	
371	btcsuite/btcd	3,868	1,308	9f0179fd2c	
372	davecgh/go-spew	3,868	246	d8f796af33	*nm*
373	FiloSottile/age	3,860	114	c9a35c0727	
374	cookieY/Yearning	3,857	1,206	4cf2b0c92b	
375	contribsys/faktory	3,857	160	1d78910fd1	
376	travisjeffery/jocko	3,857	274	06647921b3	
377	HFO4/gameboy.live	3,851	174	ddfaa9ce46	
378	gonum/gonum	3,848	306	5f68e0b712	
379	goadesign/goa	3,828	433	05b00660da	
380	schollz/find3	3,810	267	601134997e	
381	motemen/gore	3,796	133	90b551f3a0	
382	go-pg/pg	3,791	288	7aa4fb28bf	
383	elves/elvish	3,785	230	8af7608382	
384	uber-archive/go-torch	3,774	217	86f327cc82	*nm*
385	nsf/termbox-go	3,772	315	38ba6e5628	*nm*
386	sql-machine-learning/sqlf…	3,772	566	bd1d181fbc	
387	gogo/protobuf	3,771	495	5628607bb4	
388	emicklei/go-restful	3,762	576	7c2556bbbf	*nm*
389	google/wire	3,757	189	978b7803d3	
390	uber/cadence	3,753	341	a1a14830e4	
391	tj/node-prune	3,739	91	1159d4cac8	

Continued

Continued

	Name	Stars	Forks	Revision	
392	elazarl/goproxy	3,735	732	49ad98f6da	
393	mitchellh/gox	3,734	287	d8caaff5a9	
394	syndtr/goleveldb	3,728	551	758128399b	
395	satori/go.uuid	3,717	496	b2ce2384e1	*nm*
396	maxence-charriere/go-app	3,713	173	157b232cd2	
397	wercker/stern	3,709	188	4fa46dd698	*nm*
398	rgburke/grv	3,698	90	cffa246f92	*nm*
399	jpmorganchase/quorum	3,689	991	282b74045a	*nm*
400	tophubs/TopList	3,686	688	04cfb28f8a	
401	alexflint/gallium	3,674	137	29fad37751	*nm*
402	oxequa/realize	3,660	192	498ce46d1b	
403	getlantern/lantern	3,629	10,622	08fd2e3f22	*nm*
404	knadh/listmonk	3,625	169	82702ed5bc	
405	panjf2000/ants	3,624	456	a195593eb7	
406	tendermint/tendermint	3,615	1,115	75e19f99ea	
407	huichen/wukong	3,612	779	d014a1f19d	*nm*
408	kashav/fsql	3,605	105	aea2c2b0fa	*nm*
409	jwilder/docker-gen	3,596	499	4edc190faa	*nm*
410	fanpei91/torsniff	3,595	897	e81c707828	*nm*
411	zenazn/goji	3,588	242	a16712d37b	*nm*
412	sourcegraph/sourcegraph	3,588	365	32f87ec190	
413	gchaincl/httplab	3,569	119	ddfb5124d4	
414	alecthomas/gometalinter	3,560	282	32416ab753	*nm*
415	hashicorp/raft	3,557	473	c95aa91e60	
416	openark/orchestrator	3,556	573	76c0a52560	*nm*
417	jung-kurt/gofpdf	3,535	438	a2a0e7f8a2	
418	name5566/leaf	3,533	969	8592b1abbb	*nm*
419	OWASP/Go-SCP	3,531	235	e152aab8cb	*nm*
420	golang/lint	3,518	474	738671d388	
421	yuin/gopher-lua	3,503	385	6ff375d91e	
422	jenkins-x/jx	3,490	666	a3f8cdf623	
423	anacrolix/torrent	3,489	413	c7ea314de0	
424	99designs/aws-vault	3,489	302	83ed01d1d1	
425	imgproxy/imgproxy	3,485	243	3dcaeedd45	
426	googollee/go-socket.io	3,481	570	88e44eae2d	
427	operator-framework/operat…	3,477	1,002	d1dbfc8336	
428	tilt-dev/tilt	3,472	113	9b58b81e77	
429	buger/jsonparser	3,468	296	277c1bf2e4	
430	fatih/color	3,468	418	daf2830f27	
431	google/pprof	3,460	299	427632fa3b	
432	ro31337/libretaxi	3,459	799	64b7e21086	*nm*
433	gomatcha/matcha	3,456	147	d2dd1c5db1	*nm*
434	open-policy-agent/opa	3,450	404	b00f6aac6c	
435	kubesphere/kubesphere	3,450	525	ea2e87697d	
436	hoisie/web	3,448	785	a498c022b2	*nm*
437	michenriksen/aquatone	3,444	630	854a5d56fb	*nm*
438	dvyukov/go-fuzz	3,443	205	be3528f3a8	*nm*
439	google/gopacket	3,433	687	a83d5be3a7	
440	Tencent/bk-cmdb	3,428	1,121	011400903f	*nm*
441	kubernetes/client-go	3,422	1,464	f099a72e14	
442	ant0ine/go-json-rest	3,417	381	ebb33769ae	*nm*
443	mozilla-services/heka	3,408	559	278dd3d596	*nm*
444	baidu/bfe	3,405	581	5f85b4cf4b	
445	igrigorik/ga-beacon	3,404	341	3a9a02fc0f	*nm*
446	gopasspw/gopass	3,396	286	0549c36ab0	
447	letsencrypt/boulder	3,393	417	286271f0db	
448	rs/zerolog	3,386	212	7825d86337	
449	uber/kraken	3,385	205	1c0ee7ab82	
450	jwilder/dockerize	3,378	243	9ce2e80619	*nm*
451	mitchellh/mapstructure	3,374	378	20e21c67c4	
452	gorgonia/gorgonia	3,373	301	5fb5944d4a	*
453	codegangsta/gin	3,370	277	cafe2ce989	*nm*
454	manifoldco/promptui	3,369	157	e15db71081	
455	go-gorp/gorp	3,361	362	03c0a1b169	*nm*
456	ansible-semaphore/semapho…	3,360	482	3db758c066	*nm*
457	DisposaBoy/GoSublime	3,351	315	bd0163d030	*nm*
458	vugu/vugu	3,331	132	895fa1f82f	
459	heroiclabs/nakama	3,325	355	8e8e742809	
460	graph-gophers/graphql-go	3,324	350	dae41bde9e	
461	tsuru/tsuru	3,323	450	05c44a1bbf	
462	ffhelicopter/Go42	3,321	468	ad6fd3cfde	*nm*
463	burke/zeus	3,310	232	a77e410c21	*nm*
464	tinode/chat	3,309	655	8c524edf2c	
465	GoAdminGroup/go-admin	3,308	592	d2e4932b28	
466	OWASP/Amass	3,306	588	d69ce878a2	
467	variadico/noti	3,305	93	0538ed50e6	*nm*
468	ledisdb/ledisdb	3,297	377	d35789ec47	
469	gilbertchen/duplicacy	3,296	241	76f1274e13	*nm*
470	nuclio/nuclio	3,283	326	59cab8961c	
471	tianon/gosu	3,278	207	ad4ec33beb	*nm*
472	adtac/commento	3,273	144	326601394a	*nm*
473	meshbird/meshbird	3,270	199	de2c7fe497	
474	gwuhaolin/lightsocks	3,255	761	7e9e1a234b	
475	securego/gosec	3,252	245	6a130d55b3	
476	astaxie/go-best-practice	3,249	695	67aa4052f1	*nm*
477	asticode/go-astilectron	3,239	220	3961c75232	
478	hybridgroup/gocv	3,234	481	df73d54467	

Continued

Continued

	Name	Stars	Forks	Revision	
479	streadway/amqp	3,227	480	`1c71cc93ed`	
480	dominikh/go-tools	3,210	196	`1dc5519e1d`	
481	BurntSushi/toml	3,205	406	`3012a1dbe2`	*nm*
482	hasura/gitkube	3,204	154	`ec3d06703f`	*nm*
483	OJ/gobuster	3,189	549	`92ffe359d2`	
484	davyxu/cellnet	3,187	744	`a12953d6f6`	
485	bitly/go-simplejson	3,186	440	`39a59b1b28`	
486	AsynkronIT/protoactor-go	3,182	338	`c483abfa40`	
487	ha/doozerd	3,181	261	`8adec5411b`	*nm*
488	kubernetes/autoscaler	3,178	1,424	`a00bf59159`	*nm*
489	prometheus/alertmanager	3,169	1,191	`97bd078441`	
490	gobwas/ws	3,167	206	`d1714d5c97`	*nm*
491	kgretzky/evilginx2	3,164	602	`fe4e343143`	
492	fullstorydev/grpcurl	3,160	144	`44547153b3`	
493	gomods/athens	3,159	367	`28d606947a`	
494	schollz/croc	3,151	155	`0377223a67`	
495	nytimes/gizmo	3,145	205	`3868112445`	
496	disintegration/imaging	3,089	271	`879073f233`	
497	gogf/gf	3,087	450	`269378aa0d`	
498	googleforgames/agones	3,082	356	`d017250c43`	
499	kubernetes-sigs/external-…	3,078	926	`deaeca2ab1`	
500	gwuhaolin/livego	3,075	902	`bf65635cef`	

List of Abbreviations

API Application Programming Interface.

ASLR Address Space Layout Randomization.

AST Abstract Syntax Tree.

CFG Control Flow Graph.

CLI Command Line Interface.

CPU Central Processing Unit.

CSV Comma-Separated Values.

DEP Data Execution Prevention.

DNS Domain Name System.

EA Escape Analysis.

ECB Electronic Code Book.

FFI Foreign Function Interface.

FUSE Filesystem in Userspace.

GC Garbage Collector.

HTTP Hypertext Transport Protocol.

LOC Lines of Code.

RIP Return Instruction Pointer.

ROP Return Oriented Programming.

List of Figures

List of Tables

List of Listings

References

[1] Hussain M. J. Almohri and David Evans. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 248–255, New York, NY, USA, March 2018. Association for Computing Machinery.

[2] Saleh Mohamed Alnaeli, Melissa Sarnowski, Md Sayedul Aman, and Ahmed Kumar Yelamarthi. Source Code Vulnerabilities in IoT Software Systems. In *Advances in Science, Technology and Engineering Systems Journal*, volume 2 of *3*, pages 1502–1507, August 2017.

[3] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 464–467, New York, NY, USA, May 2016. Association for Computing Machinery.

[4] Amitav Banerjee, U. B. Chitnis, S. L. Jadhav, J. S. Bhawalkar, and S. Chaudhury. Hypothesis testing, type I and type II errors. *Industrial Psychiatry Journal*, 18(2):127–131, 2009.

[5] Eric Bodden, Ka I. Pun, Martin Steffen, Volker Stolz, and Anna-Katharina Wickert. Information Flow Analysis for Go. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, Lecture Notes in Computer Science, pages 431–445, Cham, 2016. Springer International Publishing.

[6] S. S. Brimzhanova, S. K. Atanov, Moldamurat Khuralay, K. S. Kobelekov, and L. G. Gagarina. Cross-platform compilation of programming language Golang for Raspberry Pi. In *Proceedings of the 5th International Conference on Engineering and MIS*, ICEMIS '19, pages 1–5, New York, NY, USA, June 2019. Association for Computing Machinery.

[7] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 381–392, New York, NY, USA, May 2018. Association for Computing Machinery.

[8] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, October 1999.

[9] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[10] Diego Elias Costa, Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. Breaking Type-Safety in Go: An Empirical Study on the Usage of the unsafe Package. *arXiv:2006.09973*

[cs], June 2020.

[11] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, March 2011. Association for Computing Machinery.

[12] Fernando López de la Mora and Sarah Nadi. Which library should I use? A metric-based comparison of software libraries. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, pages 37–40, New York, NY, USA, May 2018. Association for Computing Machinery.

[13] Nicolas Dilley and Julien Lange. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 377–387. IEEE, February 2019.

[14] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, October 2015.

[15] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84, New York, NY, USA, November 2013. Association for Computing Machinery.

[16] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is Rust Used Safely by Software Developers? In *42nd International Conference on Software Engineering (ICSE '20)*, Seoul, Republic of Korea, May 2020. Association for Computing Machinery, New York, NY, USA.

[17] Leo Ferres. Memory management in C: The heap and the stack. *Department of Computer Science, Universidad de Concepcion*, October 2010.

[18] Julia Gabet and Nobuko Yoshida. Static race detection and mutex safety and liveliness for Go programs. In *34th European Conference on Object-Oriented Programming (ECOOP) 2020, Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl Leibniz-Zentrum für Informatik, to appear.

[19] YC Gao, AM Zhou, and Liang Liu. Data-Execution Prevention Technology in Windows system. In *Information Security & Communications Privacy*. Electronic Information College, Sichuan University, Chengdu Sichuan, China, July 2013.

[20] Marco Giunti. GoPi: Compiling linear and static channels in Go. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages*, pages 137–152, Cham, 2020. Springer International Publishing.

[21] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105, New York, NY, USA, July 2015. Association for Computing Machinery.

[22] John Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, May 1998.

[23] J. Heffley and P. Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences, 2004.* IEEE, January 2004.

[24] Patricia M. Hill and Fausto Spoto. A Foundation of Escape Analysis. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 380–395, Berlin, Heidelberg, 2002. Springer.

[25] Shiyou Huang, Jianmei Guo, Sanhong Li, Xiang Li, Yumin Qi, Kingsum Chow, and Jeff Huang. SafeCheck: Safety Enhancement of Java Unsafe API. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 889–899. IEEE, May 2019.

[26] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, December 2017.

[28] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. CogniCrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, October 2017.

[29] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2018. Schloss Dagstuhl Leibniz-Zentrum für Informatik.

[30] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? In *Empirical Software Engineering (2018)*, pages 384–417, New York, May 2017. Springer Science+Business Media.

[31] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. *ACM SIGPLAN Notices*, 52(1):748–761, January 2017.

[32] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *10th USENIX Security Symposium*. USENIX Association, 2001.

[33] Johannes Lauinger, Lars Baumgärtner, Anna-Katharina Wickert, and Mira Mezini. Uncovering the Hidden Dangers: Finding Unsafe Go Code in the Wild. In *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020, Guangzhou, China, December 29, 2020 – January 1, 2021*. IEEE, to appear. Preprint available: arXiv:2010.11242.

[34] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.

[35] Hector Marco-Gisbert and Ismael Ripoll. On the Effectiveness of Full-ASLR on 64-bit Linux.

In *Proceedings of the In-Depth Security Conference (DeepSec)*, November 2014.

[36] Luis Mastrangelo. *When and How Java Developers Give Up Static Type Safety*. PhD thesis, Università della Svizzera Italiana, 2019.

[37] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The Java unsafe API in the wild. *ACM SIGPLAN Notices*, 50(10):695–710, October 2015.

[38] Nicholas D. Matsakis and Felix S. Klock. The Rust Language. *ACM SIGAda Ada Letters*, 34(3):103–104, October 2014.

[39] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 84–94, Urbana, IL, USA, October 2017. IEEE.

[40] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 41–62, Berlin, Heidelberg, 2016. Springer.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, June 2009. Association for Computing Machinery.

[42] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable Open Source Dependencies: Counting Those That Matter. In *ESEM '18: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, Oulu, Finland, October 2018. Association for Computing Machinery.

[43] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, New York, NY, USA, June 2020. Association for Computing Machinery.

[44] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.

[45] Mike Rourke. *Learn WebAssembly: Build Web Applications with Native Performance Using Wasm and C/C++*. Packt Publishing Ltd, September 2018.

[46] Andrey Sibiryov. Golangs garbage. USENIX Association, May 2017.

[47] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *USENIX Symposium on Usable Privacy and Security (SOUPS) 2020*, pages 169–184, Virtual, August 2020. USENIX Association.

[48] Nathan P Smith. Stack smashing vulnerabilities in the UNIX operating system. May 1997.

[49] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, May 2019.

[50] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, May 2013.

[51] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878, Providence, RI, USA, April 2019. Association for Computing Machinery.

[52] Matthias Vallentin. On the evolution of buffer overflows. May 2007.

[53] Cong Wang, Mingrui Zhang, Yu Jiang, Huafeng Zhang, Zhenchang Xing, and Ming Gu. Escape from Escape Analysis of Golang. In *ICSE*, Seoul, Republic of Korea, May 2020. Association for Computing Machinery.

[54] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 14–24, Buenos Aires, Argentina, May 2017. IEEE.

[55] Arthur Henry Watson, Dolores R. Wallace, and Thomas J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[56] Anna-Katharina Wickert, Michael Reif, Michael Eichberg, Anam Dodhy, and Mira Mezini. A Dataset of Parametric Cryptographic Misuses. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, March 2019.

[57] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying Reuse of Out-Dated Third-Party Code in Open Source Projects. *Information and Media Technologies*, 9(2):155–161, 2014.

[58] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 914–919, Paderborn, Germany, August 2017. Association for Computing Machinery.

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Johannes Tobias Lauinger, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, October 26, 2020

Johannes Tobias Lauinger