

Breaking Type-Safety in Go: An Empirical Study on the Usage of the `unsafe` Package

Diego Elias Costa, Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab *Senior Member, IEEE*

Abstract—A decade after its first release, the Go programming language has become a major programming language in the development landscape. While praised for its clean syntax and C-like performance, Go also contains a strong static type-system that prevents arbitrary type casting and arbitrary memory access, making the language type-safe by design. However, to give developers the possibility of implementing low-level code, Go ships with a special package called `unsafe` that offers developers a way around the type-safety of Go programs. The package gives greater flexibility to developers but comes at a higher risk of runtime errors, chances of non-portability, and the loss of compatibility guarantees for future versions of Go.

In this paper, we present the first large-scale study on the usage of the `unsafe` package in 2,438 popular Go projects. Our investigation shows that `unsafe` is used in 24% of Go projects, motivated primarily by communicating with operating systems and C code, but is also commonly used as a source of performance optimization. Developers are willing to use `unsafe` to break language specifications (e.g., string immutability) for better performance and 6% of analyzed projects that use `unsafe` perform risky pointer conversions that can lead to program crashes and unexpected behavior. Furthermore, we report a series of real issues faced by projects that use `unsafe`, from crashing errors and non-deterministic behavior to having their deployment restricted from certain popular environments. Our findings can be used to understand how and why developers break type-safety in Go, and help motivate further tools and language development that could make the usage of `unsafe` in Go even safer.

Index Terms—Go language, `unsafe`, type-safety, software packages, Empirical Study.

1 INTRODUCTION

The famous Uncle Ben’s quote “With great power comes great responsibility” is a proverb that can be aptly applied to unsafe packages or libraries¹ in programming languages. Statically-typed programming languages, such as Java, Rust, and Go restrict developer’s freedom (power) by requiring every variable and function to have an explicit type in favor of a safer environment, capturing illegal type conversions and memory accesses at compile time. While efficient at identifying type violations, the restriction to write type-safe implementations would make it impossible to write low-level functions. Language designers address this problem by including a backdoor to violate the type system in the form of unsafe packages. Java has `sun.misc.Unsafe`, Rust has `unsafe Rust`, and Go has the `unsafe` package. Unsafe packages give the much needed flexibility to write type-unsafe functions, for low-level implementations and optimizations, but need to be used with extreme care by developers [33], [47].

Go is a statically-typed and compiled programming language released over a decade ago by engineers at Google [3]. Its simple syntax and high efficiency has made Go one of the major programming languages in the current development landscape [6], [39]. Go has a strong static type system, but

ships with the `unsafe` package [33] to offer developers the possibility of implementing low-level functions. This package offers a way-around the type-safety of Go programs, but it comes with a series of important drawbacks. First, programs that use `unsafe` may not be portable to different CPU architectures, and are not guaranteed to be compatible to future versions of Go [2]. Second, some `unsafe` operations contain hidden complexities that can run the program rogue. For instance, Go contains pointers and a Garbage Collector (GC), hence, manipulating (unsafe) pointers without proper care may cause the GC to release unwanted variables [9].

Given that the benefits of writing type-safe code is well-known and lifting this safety net puts programs at a higher risk of runtime errors [47], why do developers break type-safety in Go? There is no shortage of articles in the web warning against the perils of using `unsafe` [4], [12], [33], and maintainers of Go have had extensive debates over the need and consequences of keeping this package in the language [21]. However, it is hard to derive effective measures on how to handle the risks and benefits of `unsafe` package without knowing the extent in which developers use it, why they use `unsafe` for, and what are the real risks of breaking type-safety in Go projects. Our study is a step towards acquiring this understanding.

In this paper, we perform a mix-method empirical study involving 2,438 popular Go open-source projects. We first develop a parser to identify usages of the `unsafe` package throughout the development history of Go projects. Then, we perform a manual analysis to catalogue the most common `unsafe` usages, and qualitatively evaluate the risks related to using the `unsafe` package. Our study focus on answering the following research questions:

- D. E. Costa, S. Mujahid, and E. Shihab are with the Data-driven Analysis of Software (DAS) Lab at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.
E-mail: d_damasc, s_mujahid, eshihab@encs.concordia.ca
- R. Abdalkareem is with Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen’s University, Canada.
E-mail: abdrabe@gmail.com

Manuscript received XXXXX; revised XXXXX.

1. In this paper, we use the term package to refer to a software library.

RQ1: Is `unsafe` widely used in open-source Go projects?

We found that 24% of the studied Go projects use `unsafe` at least once in their code-base. While the number of `unsafe` call-sites tends to increase as projects evolve, developers tend to keep `unsafe` usage on a proportionally reduced number of packages. The package `unsafe` is used in a wide variety of project domains (e.g., networking, development tools, databases). We also found that projects that implement bindings to other platforms and programming languages tend to rely more heavily on the `unsafe` package (more than 100 call-sites).

RQ2: Why do developers use `unsafe`?

We catalogued 6 groups of usage-patterns related to `unsafe` in Go. The majority of `unsafe` usages were motivated by integration with operating systems and C code (45.7%), but developers also frequently use `unsafe` to write more efficient Go code (23.6%). Less frequently, developers use `unsafe` to perform atomic operations, dereference values through reflection, manipulate memory addresses and get the size of objects in memory.

RQ3: What are the risks of using `unsafe`?

Approximately 6.6% of the investigated projects that use `unsafe` have invalid pointer conversions, a risky usage of the API that may cause crashing bugs and lead programs to unexpected runtime behavior. Projects that use `unsafe` report a variety of exclusive issues, from having their deployment restricted by environment platforms, to crashing and non-deterministic errors found in production. As such, we also found that developers also make the effort to reduce or remove the dependency to `unsafe` to mitigate related issues.

Our study provides empirical evidence that contributes towards a safer Go programming language. Our findings show that the usage of `unsafe` is widespread, covering all sorts of projects domains, and is motivated by integration with Operating Systems, C programs and more efficient implementations. The usage patterns we catalogued can be used by tool designers to better assist developers when performing `unsafe` operations, as well as guiding standard packages to improve documentation to cover the most common use-cases. Furthermore, our risk analysis indicates that even popular projects are not immune to well-known pitfalls associated with the package usage and that projects that use `unsafe` report a variety of `unsafe`-related issues. Our results may also help developers at identifying potential risks and pitfalls to avoid when using the `unsafe` package.

This paper is organized as follows: Section 2 introduces the `unsafe` package and the concepts we will rely upon throughout the paper. Section 3 presents the methodology used for our study. The results of our study are presented in three sections, Section 4 presents the results of RQ1, Section 5 shows RQ2 results, and RQ3 results are presented in Section 6. We discuss our findings and implications in Section 7. Then, in Section 8 we present the related work and discuss the threats to the validity of our results in Section 9. Finally, we conclude our study in Section 10.

2 BACKGROUND

In this section, we introduce the `unsafe` package API and exemplify some of its use-cases. We describe the risks of using `unsafe` and dive into a particular `unsafe`-related pitfall that may run programs into crashing bugs and non-deterministic behavior.

2.1 Type-safety in Go and the `unsafe` package

Go is a statically typed language. That is, the type of variables in a Go program are known at compile time, enabling the compiler to verify type inconsistencies and incompatible type conversions. A compiled Go program is guaranteed to be type-safe which prevents a myriad of issues to happen at runtime, unless developers use the `unsafe` package. As described in the official documentation shown in Figure 1, the `unsafe` package offers a step around the type safety of Go programs [33]. The `unsafe` package is quite compact, containing only three functions and a Pointer type in its API, shown in the Index of Figure 1-A. By using the `unsafe` package, developers have the flexibility needed to develop low-level functions, such as full control over pointers (C-style), the ability to read and write arbitrary memory, access to object sizes, and the possibility to convert between any two types with compatible memory layouts (shown in Figure 1-B).

2.2 Risks of using the `unsafe` package

In practical terms, breaking type-safety means lifting the safety net a compiler provides for developers in exchange for the full-control of reading and writing memory. This by itself, puts developers at a higher risk of making mistakes that will ripple through to the production environment. For instance, converting between incompatible types may cause the program to crash or misinterpret the memory layout of a variable causing unexpected program behavior. Aside from this, the `unsafe` package comes with a series of particular drawbacks described in the package documentation [33], that need to be taken into cautious consideration by developers:

Non-Portability: A regular package in Go can be compiled to different CPU architectures and operating systems without any changes in the code. Low-level implementations that use `unsafe`, however, may need to account for differences in the CPU architectures and it is up to developers to keep their packages portable. For instance, by traversing an array of integers using `unsafe`, developers need to account for the integer size which differs in x86 and x64 architectures. Another example is the reliance on system calls, which are not often portable to different operating systems.

Not Protected by the Compatibility Guideline: Programs that are written in Go have the guarantee to work in future versions of the programming language, as long as they follow the compatibility guideline established in Go 1 [2]. However, using the `unsafe` package breaks this compatibility guarantee, as `unsafe` exposes the internal implementations of the language, which may change in future versions. Hence, programs that rely on `unsafe` are not guaranteed to work in future implementations of Go.

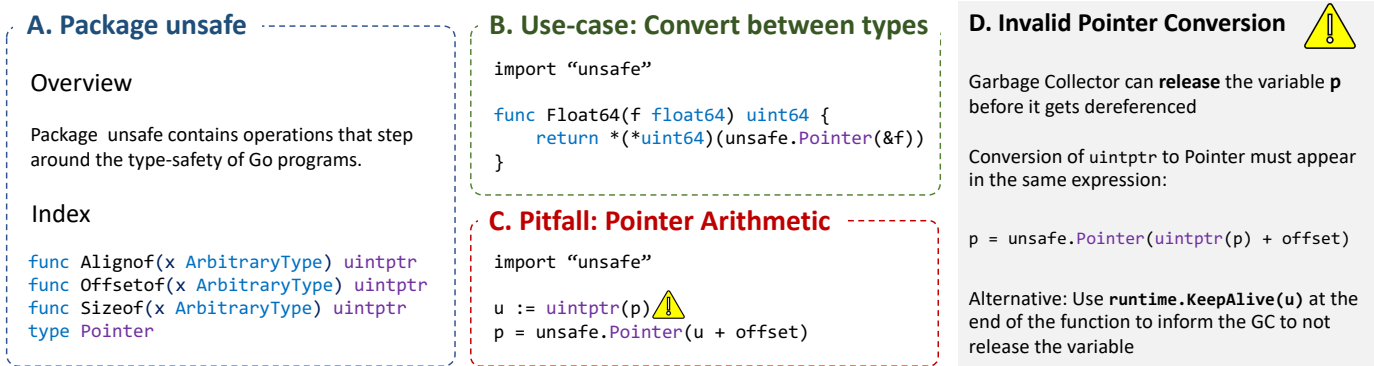


Fig. 1: The `unsafe` package API (A), an example of how to use `unsafe` to convert between types in Go (B) and a pitfall related to performing pointer arithmetic in Go (C). On the right we explain the invalid pointer conversion issue and give an example of solution for a safe pointer arithmetic.

2.3 Invalid Pointer Conversion

The `unsafe` package provides an extensive documentation on how to properly use the package and some of the pitfalls developers must avoid when breaking type-safety [33]. A pitfall that is well-described in the package documentation is the invalid pointer conversion. As a rule of thumb, converting pointer addresses back to pointer variables is not valid in Go language. Pointer memory addresses in Go are of the type `uintptr` (unsigned integer pointer), which is a simple integer type, holding no reference or pointer-semantics. Since Go has a Garbage Collector that releases memory that is not referenced by any variable in the program, a variable `uintptr` holding the address of a pointer will not prevent the garbage collector to release the said pointer. Consequently, there is no guarantee that a memory address (type `uintptr`) contains a valid Go pointer variable to be dereferenced [9].

In some cases, however, there is a valid need to manipulate memory addresses and dereference them back to pointers. For instance, to perform pointer arithmetic operations (C-style) to traverse an array as shown in Figure 1-C. Developers need to be aware of the intricacies of manipulating low-level pointers in a language that contains a Garbage Collector, to prevent the Garbage Collector from releasing their variables in the middle of their operations. We show in Figure 1-D an example on how to properly handle pointer arithmetics in Go. In this case, developers should never store their `uintptr` into an intermediate variable (`u`), because at this point in the execution of a program the variable `p` can be released by the Garbage Collector. This example shows that the pitfalls of handling `unsafe` are not always intuitive. To make matters worse, the issues that could arise from using the invalid pointer conversion are non-deterministic and are unlikely to be issued during software testing when memory pressure tends to be small.

3 METHODOLOGY

To understand the prevalence of `unsafe` and its impact in open source Go projects, our study has three main goals:

- 1) Understand the extent in which projects use `unsafe` package in their source-code (Section 4). To achieve this, we identify projects that use `unsafe` with a parser,

investigate whether the usage of the package changes as the project evolves and what are the categories of projects that more frequently rely on `unsafe`.

- 2) Understand why projects use `unsafe` (Section 5). We investigate what are the most used features from the `unsafe` API and manually extract the most common usage patterns of `unsafe` in Go projects.
- 3) Understand the risks that using `unsafe` entails to projects (Section 6). To achieve this, we examine if projects have occurrences of invalid pointer conversion in their code and qualitatively evaluate project issues related to the `unsafe` package.

We dedicated this section to describe the methodology used to achieve the goals of our study. Figure 2 presents the overview of our methodology that is detailed in the following subsections.

3.1 Study dataset

To investigate the usage of `unsafe` on a large-set of popular Go projects, we first started by querying the GitHub REST API² to identify a representative set of open source Go project. To do so, we selected a set of the top 3,000 most starred Go repositories, as the number of stars is an indicator of the project popularity within GitHub repositories [13], [14]. Our dataset was collected on October 2nd, 2019.

Even within highly starred repositories, we may find inactive repositories or repositories not related to software development. To get a representative set of high-quality and active Go software development projects, we follow the methodology recommended by previous work [41] to further curate our initial dataset through the following criteria:

- 1) We filter out 89 archived projects, as these projects are no longer maintained by the development community. Archived projects are identified via a flag in the project's metadata.
- 2) We removed 371 inactive projects, by filtering out projects that have had no contribution 12 months prior to the data collection (after October 2018).
- 3) We removed 22 projects with less than 10 commits in total, as these projects tend to be too young and are

2. <https://developer.github.com/v3/>

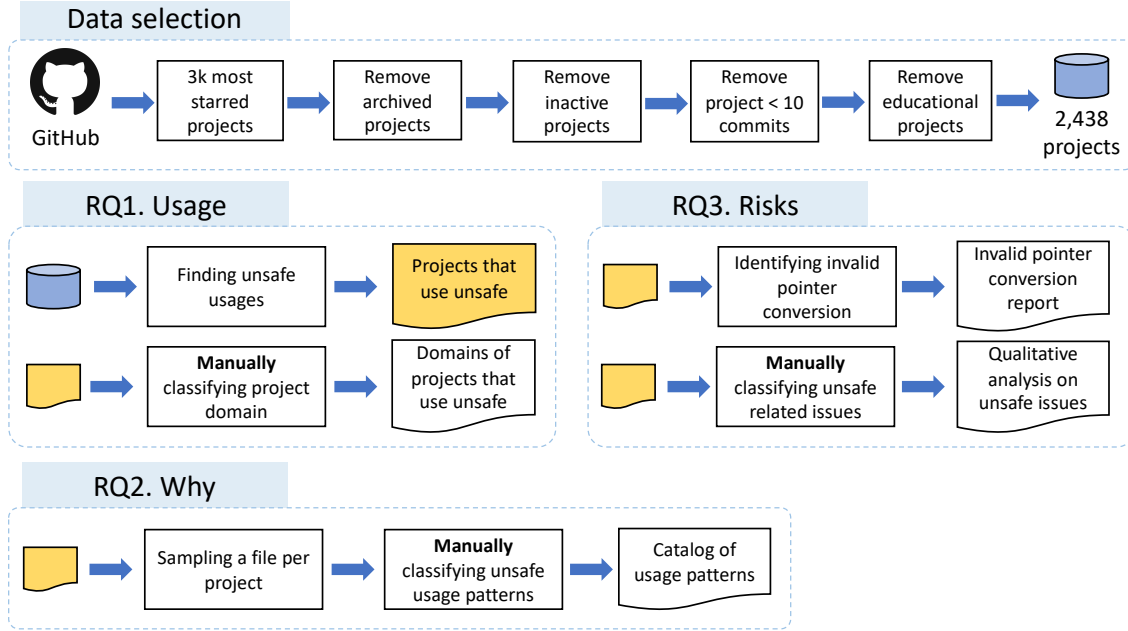


Fig. 2: Overview of the methodology adopted in our study.

TABLE 1: Statistics of the projects in our dataset.

Statistics	Mean	Min	Median	Max
Age (months)	44.17	1	43	119
Stars	1,967.81	314	858	64,079
Forks	276.71	2	109	20,437
LOC	210,217.09	77	9,606	16,579,983
Commits	1,028.48	11	261	97,504
Size (Kb)	17,793.30	6	2,476	1,151,822

not representative of the typical Go project we aim to investigate.

- 4) We also removed 56 educational repositories, which are fairly common on popular projects datasets. These projects are described as published books, programming courses or any learning material, which are invaluable to the community, but are not representative of a typical Go software. We removed these by manually inspecting the project's description of our entire dataset.
- 5) Furthermore, 24 projects could not be cloned automatically by our scripts (e.g., invalid URL, project no longer available) and were removed from our study.

This process yields a dataset composed of 2,438 Go projects. Table 1 shows the summary statistic of the selected Go projects in our dataset. As shown in Table 1, our dataset contains very popular projects (median of 857 stars and 110 forks), with a relatively long development history (median of 3.5 years of development).

3.2 Identifying unsafe usages

To find whether projects make use of `unsafe` in their source-code, we build a parser for Go source code, using the support of Go's native `ast` package³. Our parser first analyzes the source code of each Go file (files with `.go`

extension) in the selected projects and build an abstract syntax tree (AST) for each file. Then, the parser inspects the AST of Go files and identifies the function calls and type references to `unsafe`. First we run our parser at the latest snapshot of the projects obtained during our data collection. In addition, we also want to understand how the usage of `unsafe` changes as the project evolves. To that aim, we use our parser to analyze the snapshot of the first commit of each month in the project's history. For this analysis, we only consider the default branch of each project, identified via the GitHub API.

3.3 Classifying projects' domain

To complement the analysis on the usage of `unsafe`, we investigate the domain of the projects that use the `unsafe` package, as the domain may have a direct influence on the need for breaking type-safety. Intuitively, we expect projects that demand low-level implementation and optimizations, such as databases, and file systems, to depend more on `unsafe` implementations than other type of projects such as data structure libraries and web applications. To identify the projects' domain, we manually inspect the description and documentation in their GitHub page and classify each project into a dominant domain, e.g., database, compiler, web server, development tool. The three-first authors classify the repositories using an open card-sort method [25], where labels are created during the labeling process and each new label is discussed among annotators and retroactively applied to previously classified projects. When different labels were assigned to the same project we discuss to reach a consensus.

3.4 Classifying unsafe usage patterns

To investigate why developers use `unsafe` in RQ2, we need to understand the most frequent usage patterns associated

3. <https://golang.org/pkg/go/ast/>

with the `unsafe` package. Each usage pattern may offer a rationale that will help us understand why developers opted for breaking-type safety to achieve certain functionality. To extract high-level usage patterns from the projects' source code we resort to in-depth manual analysis of the code, documentation, and commit messages. This analysis is very time consuming, as annotators need to recognize the context in which `unsafe` is being used, search for clues that indicate the reason behind the `unsafe` in commit messages and code documentation. Hence, we decide to perform such analysis on a statistically significant random sample of the projects that use `unsafe`. This sample is drafted to provide us with a representative set of the projects that rely on `unsafe` with 5% confidence interval at 95% confidence level.

We expect some projects to contain hundreds of `unsafe` call-sites while most projects may use `unsafe` sporadically in their code, given the risks associated with the package. We want to identify patterns across projects and avoid biasing our analysis towards projects that rely more extensively on `unsafe` in their implementation. Hence, we perform a second sampling by randomly selecting a single file from each project to our analysis.

It is important to highlight that we opt to conduct this particular analysis at the file level as opposed to package-level for the following reasons: 1) Files are more fine-grained than packages and are expected to have a more cohesive structure where we could more easily derive the usage pattern; 2) We can analyze the context of usage of a single file, with support of `gitblame` to further inspect commit messages, without the need of inspecting method calls across different files, which would impose a prohibitive time cost to this analysis.

The first two authors independently labeled each file (one per project) using an open card-sort method [25]. Hence, similarly to the analysis of projects' domains, labels are created and assigned while inspecting the usage in source-code and git commits, and every new label is discussed among annotators and if necessary, retroactively applied to previously labelled usages. We assess the agreement of both annotators using the Cohen-Kappa metric [49] and both annotators discussed and merge the results.

3.5 Identifying invalid pointer conversions

In RQ3, we investigate the risks associated with using the `unsafe` package entail to software projects. As discussed in Section 2, a major pitfall of using `unsafe` pointers is being unaware of issues related to invalid pointer conversions. This issue can lead to non-deterministic errors in programs, as the Garbage Collector may release unintended variables and causing the program to crash. We want to investigate whether invalid pointer conversions do occur in the most popular Go projects. To automatically identify suspicious cases of invalid pointer conversions, we resort to existing static analyzers, as this issue is well-documented and covered by a native go code analyser: Vet [5]. The tool Go Vet identifies suspicious cases of invalid pointer conversion by parsing the code and applying heuristics based on ill-formed expressions, exporting every suspicious case as a json file.

3.6 Classifying unsafe related issues

Another method for understanding the risks associated with `unsafe` usage is to investigate the issues open in GitHub associated with the package use. To analyze and classify the issues related to `unsafe`, we first mine all issues from the repository of all projects that use `unsafe`. In GitHub, "issue" is an umbrella term that encloses pull requests, bugs, questions to maintainers and requests for new features. We find issues that can be candidate for our analysis by applying a keyword search for "unsafe" and its variation such as "un-safe" in the issue title. The keyword approach is prone to false-positives, especially given that the word "unsafe" is used in several different contexts (e.g., multi-threading), hence, we need to filter out false-positives from this candidate set. The first author manually inspected each issue and removed the false-positives. Then the first two authors proceed to analyze the issue title, body and related commit code to group issues based on their similarities. Similarly to the previously described methods, we resort to the open-card methodology [25] and evaluate the interrater agreement using the Cohen-Kappa interrater method [49]. We discuss the disagreement in a second round to reach a consensus in the classification. Note that the goal of this analysis is not to find all issues related to `unsafe`, but rather to classify a sample of possible `unsafe`-related issues to provide qualitative insights about the problem related to use the `unsafe` package.

3.7 Replication Package

To facilitate verification and advancement of research in the field, we provide a replication package containing the list of projects analyzed, all relevant data extracted from the project repositories, and the scripts used to process and analyze each of our RQs⁴.

4 IS UNSAFE WIDELY USED IN OPEN-SOURCE GO PROJECTS?

Motivation: We start the study by investigating to what extent developers use `unsafe` in open-source Go projects. We analyze this question under three complementary aspects:

- 1) **Usage:** How often does a Go project uses `unsafe` in their source-code? This will help us understand how frequently developers abandon type safety guarantees to implement their programs.
- 2) **Trend:** Does the usage of `unsafe` change over the evolution of a project? This analysis will give us insights on whether `unsafe` usage increases and spreads to multiple packages as the project evolves, or if developers make the conscious effort of isolating `unsafe` to mitigate its risks.
- 3) **Domain:** What domain of projects rely on `unsafe`? With this analysis, we aim at identifying what categories of projects are more susceptible to breaking type-safety.

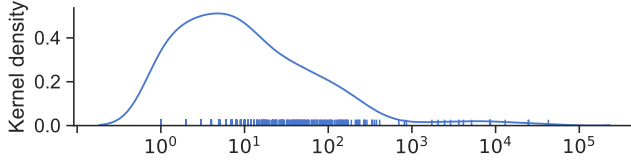
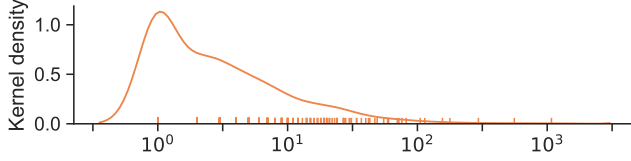
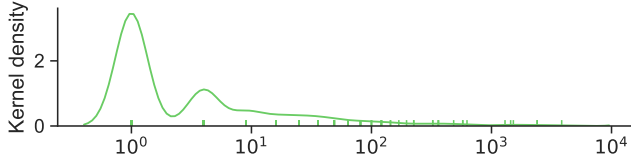
4.1 How often does a Go project use unsafe?

Approach: We run our parser to identify `unsafe` usages in every Go file present at the latest snapshot of projects in our

4. <https://zenodo.org/record/3871931>

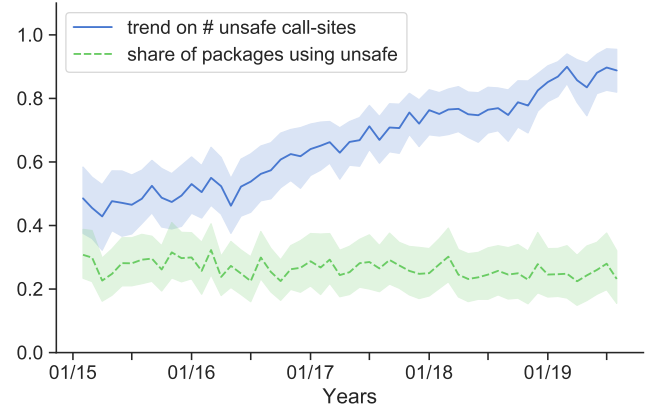
TABLE 2: Statistics on the projects that use `unsafe`.

Statistics	#	%
All projects	2,438	100.00
Projects that use <code>unsafe</code>	592	24.28
Projects with > 100 <code>unsafe</code> call-sites	69	2.83

(a) # of `unsafe` call-sites.(b) # of files with `unsafe` call-sites.(c) # of packages that depend on `unsafe`.Fig. 3: Distribution of `unsafe` usage in three granularity levels (call-site, files and packages) per project as a KDE density estimation plot. Note that the x-axis is logarithmic.

dataset, but filter out usages identified in the source-code of the project dependencies. A common practice in Go projects for managing dependencies is to include all dependencies in the projects structure, in a folder called "vendor". Hence, we exclude all reports of `unsafe` usage originating from vendor folders.

Results: Table 2 shows that from the 2,438 evaluated Go projects, **592 (24%) make use of `unsafe` directly in the project source-code**. The extent in which projects use `unsafe` varies considerably. In Figure 3 we show the distribution of the `unsafe` usage in our dataset as a Kernel-density estimation plot, under three granularities: call-sites, files, and packages. The call-sites plot accounts for every `unsafe` operation called in a project, the files indicate how many files depend on `unsafe` and the package plot shows how many modules in Go depend on `unsafe`. As evidenced by the Figure 3a peak, the majority of projects (57%) contains less than 10 calls to `unsafe` operations in their source-code. Consequently, most projects concentrate their `unsafe` calls in at most 2 files and a single package, keeping the `unsafe` usage well-localized in the code (see the distribution shown in Figures 3b and 3c). Yet, we found that 69 projects (2.8%) in our dataset rely heavily on `unsafe` in their project, with more than 100 call-sites present in their source-code.

Fig. 4: Analysis of the evolution of the number of `unsafe` operation calls in the source-code and the share of packages that depend on `unsafe` of 270 projects. We normalize the # of `unsafe` operations by the highest number of `unsafe` operations observed within a project.

4.2 Does the usage of `unsafe` change as projects evolve?

Approach: Since we want to analyze whether and how the usage of `unsafe` changed over the course of the project development, we examine the use of `unsafe` package at different snapshot of the projects history. As our dataset contains in median projects with almost 4 years of development (see Table 1), we focus on analyzing the trend of `unsafe` usage on the period between January 2015 to September 2019 (a month before our data collection time). In addition, to perform a sound analysis of the usage of `unsafe` over the years, we only conduct this analysis on projects that fulfill the following criteria: 1) projects that have at least 10 `unsafe` operation calls at their latest snapshot of their source-code, as these will show a more meaningful evolution of `unsafe` usage; and 2) projects that were being actively developed during this entire period, to avoid skewing our results towards a particular time frame. For instance, younger projects could skew our analysis as a higher number of projects would be accounted for in the most recent years.

Results: Figure 4 shows the evolution of `unsafe` of 270 projects under two perspectives: the trend on the number of `unsafe` operation calls and the percentage of packages that depend on `unsafe` in a project. The thick line represents the mean value and the colored area shows the 95% confidence interval of the data at each month. In the blue trend, we observe that the number of `unsafe` call-sites doubled in average per project, from 2015 where it had 50% of the `unsafe` call-sites to reach the maximum in 2019 (100%). Our analysis on 270 projects shows that, on average, **projects have doubled the number of `unsafe` call-sites in four years**, however the **percentage of packages that depend on `unsafe` kept steadily on 20% over time** (green trend). This shows that, as projects grow, developers are making the conscious effort of keeping `unsafe` usage concentrated on a proportionally reduced number of packages, approximately 1 every 5 packages in the 270 projects analyzed.

4.3 What domain of projects rely the most on unsafe?

Approach: We want to understand whether there is a difference in the domain of projects that use `unsafe` sporadically, against projects that rely more heavily on breaking type-safety. Hence, after categorizing the projects' domain, we group our analysis into two groups: in the first group we include all projects that use `unsafe`, and in the second group we focus on projects that rely heavily on `unsafe`, i.e., the 69 projects containing more than 100 `unsafe` call-sites.

Results: Figure 5 shows the distribution of project domains on all projects that use `unsafe` (left) and on projects that contain more than 100 `unsafe` call-sites (right). As our results show, **projects from 20 different domains make use of `unsafe`**, the five most frequent project domains that use `unsafe` package are Networking/Messaging, Development Tools, Database/Storage, Container/Virtual Machine, and Binding projects. Our results indicate that breaking type-safety is not exclusive to a handful of project domains and is employed in a variety of project categories.

Most notably, **projects that rely heavily on `unsafe` (more than 100 call-sites) were more frequently found in the Bindings domain** than in other categories. Binding projects are projects that aim at bridging the Go language to libraries and platforms written in different programming languages. These projects, such as `Gtk3` [34] a bindings for the Graphical Interface framework `GTK` [26], integrate with platforms not written in Go and use `unsafe` to implement functions that communicate with operating systems and C code. Aside from Bindings projects, other domains have a handful of projects with more than 100 `unsafe` call-sites, such as Networking (7 projects) and Database (6 projects) domains. Example of Networking and Database projects that rely heavily on `unsafe` are the Networking and Security service `Cilium` [51] and the cloud-native SQL database `CockroachDB` [18]. Interestingly, only two domain do not have projects with more than 100 `unsafe` call-sites, which are Data structures and Blockchain.

Summary of RQ1. Almost a quarter (24.28%) of most popular Go projects in our dataset use `unsafe` directly. While projects tend to increase the number of `unsafe` call-sites over time, developers tend to concentrate their usage on 20% of packages. Project from several domains use `unsafe` to some extent, but Binding projects stand out as the ones that more frequently rely heavily on `unsafe` operations.

5 WHY DO DEVELOPERS USE UNSAFE?

Motivation. Thus far, we show that almost a quarter of projects in our dataset make use of `unsafe` directly in their code, which bears the follow-up question: why developers decide to risk implementing type-unsafe routines? We want to understand the circumstances that lead developers to breaking type-safety and analyse this question under two aspects:

- 1) **Features:** What are the most used `unsafe` features? We want to identify the `unsafe` type and functions most frequently used in practice.

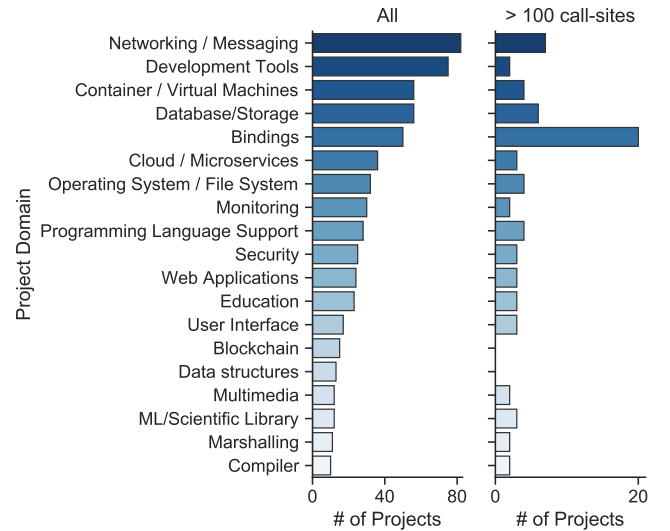


Fig. 5: Distribution of projects that rely on `unsafe` per project domain. We present the overall distribution (All) and the distribution of the 69 projects that use `unsafe` in more than 100 call-sites.

- 2) **Usage Patterns:** What developers use `unsafe` for? In this aspect, we focus on identifying and understanding the high-level usage patterns most commonly adopted in the most popular Go projects.

5.1 What are the most used unsafe features?

Approach. The `unsafe` package is very compact, composed of one type definition (`Pointer`) and three exported functions (`SizeOf()`, `OffsetOf()`, `AlignOf()`). To analyse the features used by developers, we group the analysis of `unsafe` usage conducted in the previous question by the four API components.

Results. As shown in Table 3, the `unsafe.Pointer` is the most used feature from the `unsafe` API, used by 96% of projects that rely on `unsafe` and making the bulk of 97% of all `unsafe` operation call-sites in our dataset. The `unsafe` functions are used less frequently, representing together just 2.24% of all `unsafe` operations call-sites, but are still used by a considerable number of projects. The function `Sizeof`, used to retrieve the size of a variable type, is used by 36% of all projects that rely on `unsafe`.















TABLE 3: The `unsafe` operations ranked by their usage in 592 Go projects.

Operations	Projects		Call-sites	
	#	%	#	%
type <code>Pointer</code>	570	96.28	177,192	97.85
function <code>Sizeof</code>	216	36.49	3,527	1.95
function <code>Offsetof</code>	21	3.55	312	0.17
function <code>Alignof</code>	11	1.86	43	0.02

5.2 What developers use unsafe for?

Approach: Now, we proceed to understand the usage patterns that permeates the `unsafe` usage in Go projects. From

TABLE 4: Usage patterns of `unsafe`, manually identified in 270 project files.

Usage Patterns	Description	Frequency
<code>unsafe</code>		
— Communication	Communicate with platforms and programs not written in Go	45.70% 
— System Calls	Using <code>unsafe</code> to send parameters to system calls	27.72% 
— CGO	Using <code>unsafe</code> to integrate with C code via <code>cgo</code>	17.98% 
— Efficient Casting	Perform more efficient type/array casting	23.60% 
— Type Casting	Bypassing type-checking and memory copy for performance	19.48% 
— Marshaling	Using <code>unsafe</code> to efficiently (de-)serialize json files	4.12% 
— Reflection	Access object's metadata	3.75% 
— Atomic operations	Writing atomic operations with <code>atomic/sync</code>	3.37% 
— Address Manipulation	Using <code>unsafe</code> to get memory addresses and copy memory	3.37% 
— Pointer Arithmetics	Perform pointer address arithmetics	2.25% 
— Size of Object	Getting object size in memory	6.74% 
— Getting Architecture Info	Inspecting the cpu architecture through <code>sizeof(int)</code> method	4.49% 
— Unclear	Unclear usage-patterns	3.00% 
— Others	Other use-cases (project specific)	8.24% 

the 598 projects that use `unsafe`, we randomly select 270 to perform the manual analysis (see Section 3.4). This number of projects provides us with a representative sample of the 598 projects that rely on `unsafe` with 5% confidence interval at 95% confidence level. The first two authors labeled one file per project with its dominant use-case.

Our manually analysis yielded the hierarchical label structure shown in Table 4, main six major groups of usage patterns. We aimed to be as specific as possible in our labeling, so if a *Communication* use-case can be attributed to a more specific category, say *System Calls*, we label the file usage as dominated by *System Calls*. In the cases where the specific category could not be identified, we label them as their respective super category. If neither the code of the commit messages provide sufficient information on the reason of the `unsafe` usage, we labeled the file as *Unclear*. We evaluate the labeling agreement with the Cohen-kappa interrater [49]. Cohen-kappa inter-rater is a well-known statistical method that evaluates the inter-rater agreement level for categorical scales. The result is a scale that ranges between -1.0 and 1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement. In our analysis, we found that both authors have substantial agreement ($\kappa=0.65$). The annotators discussed the divergencies to reach consensus.

Results: Our findings, depicted in Table 4, show that **developers mostly use `unsafe` for Communication to routines not written in Go (45.70%) and to perform more efficient type casting in their programs (23.60%)**. Other patterns appear less common, such as Inspecting the size of objects in memory (6.74%), using `unsafe` to inspect object's metadata through reflection (3.75%), performing atomic operations (3.37%) and general memory manipulation (3.37%). In 3% of the cases, we could not pinpoint a usage pattern based on the analysis of a single file, and in 8.24% of the cases we deemed the usage pattern too project specific to be discussed here. In the next paragraphs, we dive in details on each group of `unsafe` pattern and explain, with examples, the likely rationale behind the decision of breaking type-safety. **Communication (System Call + CGO).** The most common

Listing 1: Example of using `unsafe` to set the name of a process

```
// Unsafe pointer as a reference to a target name
ptr := unsafe.Pointer(&name_in_bytes[0])

// Setting a process name with the pointer ptr
_, _, errno := syscall.RawSyscall6(syscall.SYS_PRCTL,
    syscall.PR_SET_NAME, uintptr(ptr), 0, 0, 0, 0)
```

Listing 2: Example of using `unsafe` to call a function in C.

```
func SetIcon(iconBytes []byte) {
    // Convert to a C char type
    cstr := (*C.char)(unsafe.Pointer(&iconBytes[0]))
    // Call the function from systray.h
    C.setIcon(cstr, (C.int)(len(iconBytes)))
}
```

usage pattern in our sample of 270 projects' files is the usage of `unsafe` as a mean to communicate to routines outside of Go language. The communication to systems and programs outside of Go language requires developers to specify the memory address in which such programs can read and write, to specify parameters and receive their returned objects. For system calls, the `syscall` package [32] offers an API to different operating systems and often requires a `uintptr` with the address of a Go variable as parameters, as illustrated in Listing 1.

In turn, the package `cgo` offers a similar set of API for developers that need to integrate with C code [30]. Similarly to the `syscall` use-case, programs that call C code need `unsafe` to write and read arbitrary memory and communicate with C code, as illustrated in Listing 2. Furthermore, developers also cannot rely on the Garbage Collector from Go to release their C variables, and need to explicitly call the `C.free()` function to release the memory back to the system.

Efficient Casting. In 23.60% of the cases, developers use `unsafe` as a method of bypassing compiler checks and memory copy when casting a variable to a different type. The most common case of performance optimization is related to converting string to bytes and vice-versa. Strings

Listing 3: Example of unsafe conversion of bytes to string.

```
func String2Bytes(s string) []byte {
    sh := (*reflect.StringHeader)(unsafe.Pointer(&s))
    bh := reflect.SliceHeader{
        Data: sh.Data,
        Len:   sh.Len,
        Cap:   sh.Len,
    }
    return *(*[]byte)(unsafe.Pointer(&bh))
}
```

Listing 4: Code snippet of using unsafe to perform atomic operations, taken from the project Video-Transcoding-API.

```
// atomically set the value to avoid data races
// should probably take a different approach?
atomic.StorePointer((*unsafe.Pointer)(unsafe.Pointer(
    &c.getPresetCalledWith)), unsafe.Pointer(input.Name))
```

are immutable in Go, hence a regular type casting from a string variable requires copying the variable before the cast. To bypass this variable copy, developers use `unsafe` to change the representation of a string into a slice of bytes as illustrated in Listing 3. Since a slice of bytes can be mutated, this operation breaks the immutability of Strings as specified by the Go language, which can have far reaching consequences.

Another particular use-case for `unsafe` under this category is the more efficient marshaling functions, which accounts for 4.12% of the use-cases. The standard marshaling functions are general-purpose and use reflection to identify the object to be marshalled, an operation that can be considered slow in performance critical applications. By using `unsafe`, developers are able to implement their own customized and more efficient marshaling functions.

Reflection. Reflection allows developers to inspect and modify the metadata of types at runtime, simulating some of the dynamism of dynamic typed languages. The `reflect` package [31] requires developers to import `unsafe` to dereference an object accessed through reflection to a pointer. This is done by design, to enforce developers to import `unsafe` when performing such unsafe operations and to prevent `reflect` from replicating some of the functionality of the `unsafe` package.

Atomic Operations Another common use-case is related to the package `sync/atomic` [29]. The package provides low-level atomic functions for synchronization algorithms. As the documentation of the package poses, using `sync/atomic` properly requires great care from developers to be used correctly. This package offers a way to perform atomic operations with high-performance and without any memory allocation. Currently, there is no safe alternative to perform compare and swap operations on Go objects without memory allocation [45], which explains why the package gives support to `unsafe.Pointer` as opposed to a type-safe alternative (e.g., `interface`).

Memory Manipulation. In 3.37% of the use-cases, developers use `unsafe` to get the memory addresses and perform some arithmetic function. Most of the cases we identified are related to using the memory address as a component of a hash function, or returning the memory address as the hash of an object, similarly to the way Java implements the

Listing 5: Code snippet of using `unsafe` as a method to generate hash key, taken from the project Olric.

```
func (db *Olric) getHKey(name, key string) uint64 {
    tmp := name + key
    return db.hasher.Sum64(*(*[]byte)(unsafe.Pointer(&tmp)))
}
```

Listing 6: Code snippet of using `unsafe` to infer the system architecture, taken from the project Telegraf.

```
// Verifying the size of an integer i
if unsafe.Sizeof(i) == 4 {
    is32Bit = true
} else {
    is32Bit = false
}
```

object `hashCode()` mechanism.

Getting Object Size in Memory. The `unsafe` package can also be used to retrieve information about the object size, a use case that is often performed to get the architecture of the underlying system, such as CPU architecture and the system indianess. For instance, in Listing 6 we present a snippet where developers verify the size of an integer `i` to infer whether the operating system is 32-bits or not.

Summary of RQ2: The bulk of `unsafe` usages are related to low-level routines that communicate to operating systems and C code (45.70%), and to improve the performance of type casting (23.60%). Less frequently, developers use `unsafe` to inspect the CPU architecture (4.49%), inspect object's metadata at runtime (3.75%), perform atomic operations (3.37%) and to manipulate memory addresses (3.37%).

6 WHAT ARE THE RISKS OF USING UNSAFE?

Motivation: In previous RQs, we identified that `unsafe` is commonly used in Go projects and there are six main reasons that motivate developers to break type-safety. While using `unsafe` is dangerous by definition, we want to investigate the potential risks that using this package entails to software projects in terms of software issues. Therefore, we investigate the risks of using `unsafe` under the following two aspects:

- 1) **Invalid usage:** We analyze the `unsafe` call-sites with static analyzers to identify invalid pointer conversions in the projects source code. Invalid pointer conversion is a well-known pitfall that projects that depend on `unsafe` are at the risk of falling into.
- 2) **Unsafe-related issues:** We extract and classify issues related to `unsafe` from the projects repositories. This analysis will give us in-depth insights of real problems faced by projects that use `unsafe`.

6.1 How common is invalid pointer conversion in projects that use `unsafe`?

Approach: We run the tool Go Vet on all projects that use `unsafe`. We were able to automatically analyze 221 projects,

TABLE 5: Number of projects with invalid pointer conversions. We did not observe projects with a mix of false-positive and true-positive invalid pointer conversions.

Statistics	#	%
All projects analyzed	221	100.0%
Projects with invalid pointer conversions	14	6.3%
Projects with reported false-positives	2	0.9%

due to problems during the project build. Several projects use `unsafe` to integrate with external systems (C code and Operating Systems), hence, these projects in our dataset need external dependencies which cannot be automatically resolved with our building process. Once, we run the tool Go Vet, we count the number of reported cases of invalid pointer.

Results: Regarding invalid usage of `unsafe`, initially the Go Vet tool reported invalid pointer conversions in 16 out of 221 analyzed projects (Table 5). Upon close inspection, we found that cases found in two projects to be considered false-positives, as the conversion to `uintptr` and back to pointer occurred in the same expression, as indicated by the official `unsafe` documentation [33]. Therefore, **14 out of 221 projects (6.6%) had clear invalid pointer conversions**, with functions receiving a `uintptr` as parameter and converting it back to pointer inside the function or performing `unsafe` pointer arithmetics. Albeit occurring in a minority of the investigated projects, this analysis show that even popular projects struggle with writing `unsafe` code that is free of even the most well-documented bugs. We note that these results should be interpreted as a lower bound of invalid point conversions in these 211 projects, as there is no way to assess in reality how many of such cases were not retrieved by the tool (false negatives).

6.2 What kind of unsafe-related issues projects that use `unsafe` have?

Approach: After filtering issues through the keyword search in the title (using terms like “`unsafe`” and “`un-safe`”), we identify 286 issues from 119 projects. Our manual inspection revealed that only 103 `unsafe`-related issues from 63 projects were in fact, related to the `unsafe` package. We conduct our manual analysis on this set. Again, we use Cohen-kappa inter-rater [49] to evaluate the labeling agreement between the two annotators. We found that both annotators had a moderate agreement (Cohen-kappa=0.55) after the first labelling round. Later, both annotators discussed all divergences and reached a consensus in the classification.

Our manual classification of the `unsafe`-related issues yields a scheme shown in Table 6. We group the issues into ten categories comprised of Bug Fixing and Project Maintenance issues. We consider as Bug Fixing the issues that were open due to runtime errors or bugs found in the project code base. Project maintenance, on the other hand, are issues created with the goal of improving the project, by adding new features or refactoring the code to improve the overall quality and reduce maintenance costs. In the next paragraphs we discuss, with examples, each category of `unsafe`-related issues identified in our dataset of the 63

projects.

Unsafe Restriction. The most common `unsafe`-related issue in our dataset, found in 20 projects, is related to external environmental restriction of the use of the `unsafe` package. In most cases, this is related to the Google App Engine, a platform for cloud development that restricts the usage of `unsafe` for any Go code running in the platform [17], due to safety reasons. For example, a developer in the project GJSON wrote: “*I wanted to use this package within a Google App Engine project, and due to package “unsafe” being used, it is not compatible*” [37]. In many of these cases, the solution found for project maintainers was to provide a version of their package without `unsafe` dependency, or to remove the `unsafe` dependency in favor of a safer alternative. However, we also found cases where the usage of `unsafe` is widespread in the project’s code, this maintainers are not willing to remove it. In such cases, there is an encouragement that users fork the project repository to create a new safe-version of their packages, e.g., this developer wrote: “*The atomic swap is used all over the place during transactions, so I don’t think we’d want to take a change that removes it, but you could make a fork and replace unsafe with a mutex basically to make it less performant but safe*” [40].

Runtime Errors. In 16 projects, we encounter issues that were created due to runtime errors caused by the misuse of `unsafe` package. The most common type of runtime error, found in 6 projects, is related to crashing errors due to bad pointers, e.g. “*Prometheus crashes and hangs on ‘fatal error: found bad pointer in Go heap’*” [16]. Such errors can be caused by the Garbage Collector releasing unintended variables, mismanagement of operations that read and write memory and possibly other causes. Another cause of crashing bugs in some projects were related to the conversion between different type layouts, as a developer points out in an issue “*I suspect the problem is that there is no guarantee that the alignment is correct after the pointer conversion*” [28]. The misuse of `unsafe` have also reported to cause data corruption, “*Unsafe use of unsafe that leads to data corruption*” [52] due breaking string immutability in the string to bytes conversion. Furthermore, we also found reports indicating that the program did not crash during execution but the wrong usage of `unsafe` but has led the program to produce wrong results, as a developer mentions in an issue: “*The combination of this version of siphash and use of unsafe.Pointer to obtain a byte slices caused back-to-back ast#Term.Hash calls to return different values!*” [53]. These errors are difficult to diagnose and replicate, 5 of such issues mentioned problems to reproduce the runtime error due to the non-deterministic nature of the problem, as a developer describes: “*But someday, when using my JS package, I stumbled upon an unexpected behavior in one Lua function⁵ it is not even an error, just that the string.gsub function isn’t behaving correctly*” [24]. All such reports corroborate with the expected risks of breaking type-safety, which may cause crashing errors, data corruption, wrong behavior and in many cases are difficult to diagnose and replicate.

Wrong Usage. We group in this category, 14 bugs found

5. Upon inspection, maintainers described that the particular Lua function is implemented in Go, through the gopher-lua package.

TABLE 6: Issues related to `unsafe` identified in 63 projects. We present both the number of issues (#) and the number of projects in which they occur (# proj.).

Task	Issue category	Reason behind the Issue	#	# Projects
Bug Fixing	Unsafe restriction	Deployment environment restricts the use of <code>unsafe</code>	30	20
	Runtime errors	Wrong usage of <code>unsafe</code> caused runtime and chashing errors	18	16
	Wrong usage	Bug found in the code related to wrong usage of <code>unsafe</code> (preemptive)	14	12
	Static Check Violations	Bug in the code found by static code analyzers (Go Vet or Go1.14)	7	6
	Breaking Changes	Issue due to breaking changes introduced in <code>unsafe</code>	4	3
	Portability Issues	Program did not work in different architectures	3	2
Maintenance	Remove <code>unsafe</code>	Replace <code>unsafe</code> with a safer implementation variant	13	12
	Extending <code>unsafe</code> support	Add support to <code>unsafe.Pointer</code>	6	6
	Using <code>unsafe</code> for optimization	Optimize code using <code>unsafe</code>	6	4
	Isolate <code>unsafe</code> usage	Move <code>unsafe</code> code to a dedicated package	2	2

on 12 projects caught by maintainers or collaborators while inspecting the code, with no report of runtime issues. The most common case, found in 5 projects, is related to slice conversion: *“While not likely to occur in the wild, changes to how GC inlines functions in 1.12 creates the possibility for data loss when serialization. littleendian.go uses unsafe to change the type of slices”* [43]. Slice conversions also suffer from invalid pointer conversion issues, having the Garbage Collector release the slice in the midst of the conversion. A common employed solution for this problem is to explicitly inform the runtime system to not release the slice header during the conversion, by calling the function `runtime.KeepAlive` as illustrated in Figure 1 in the Background section. We also found issues related to traditional invalid pointer conversion, as described by a developer in an issue: *“The following code is invalid because it puts a non-pointer value into a pointer-type, if the GC finds this non-pointer it will crash the program”* [10]. Furthermore, a particular issue related to `unsafe` usage draw our attention. The issue was reported as a security related in the project `nuclio`, where conversion from string to bytes raised the possibility of exposing parameters from an internal library, as a developer reports: *“Without going through their code this [issue] is likely due to buffer reuse, which is a race condition at the least and a security issue at the worst”* [57].

Static-Check Violation. In total, 7 issues from 6 projects were opened due to static checkers identifying suspicious usage of `unsafe`. In 5 out of 7 cases, the issue was raised by the newest Go 1.14 version (released in February, 2020) which incorporated in the compiler a more robust invalid pointer conversion checker, that instruments the code to find violation to the `unsafe` rules dynamically. For example, a developer in the project `GopherLua` reported: *“Go 1.14 introduces new runtime pointer checking, enabled by default in race mode which verify that the rules for using unsafe are followed”* [38]. This check is performed if the program is compiled with a race detector (flag `-race`) in Go 1.14 and emits fatal error if a violation is found, forcing developers to open issues and fix the problem. The other two cases were motivated by `go vet`, the static analyzer we used to identify invalid pointer conversions in our previous analysis.

Breaking Changes and Portability Issues. While less frequent, we also found issues related to breaking changes (3 projects), and related to portability issues (2 project). For instance, a change in the function `sizeof` in 2011 of the package `unsafe`, created several bugs in a project. As a

developer describes *“In tip, unsafe.Sizeof has been changed to return uintptr. This causes a variety of issues during the build of walk”* [55]. We also found portability issues related to projects that use `unsafe` when used in different Operating Systems. For instance, in one issue a developer reported that *“unsafe (marshaler and unmarshaler) test failures on big endian architectures”* [56], indicating that their `unsafe` marshaller implementation worked on powerpc (32-bit) but did not work on s390x (64-bit) architectures. These issues corroborate with the `unsafe` package documentation, which clearly states that portability and compatibility guarantees are lifted off if developers use `unsafe`.

Project Maintenance. We report Project Maintenance issues related to `unsafe` grouped in four groups. Such issues are not really bugs found in projects and do not represent a direct risk for the projects. Instead, they represent tasks that are open by collaborators to improve project maintainability and are reported here in our study for completeness. The most common issue, found in 12 projects, were created to suggest the removal of `unsafe` to mitigate the risks of using `unsafe`, *“Our benchmarks also show that there is no significant difference between safe and unsafe. This allows to remove optimizations with unsafe and simply rely on plain code generation”*. [54]. Moreover, we found 2 projects with issues related to isolating the `unsafe` dependency to a reduced number of packages in a project. This indicates that developers are keen to reduce the reliance on `unsafe` whenever possible due to its inherent risks. While there is an effort for keeping the `unsafe` usage at its minimal in some projects, we also found that developers expand their API to support the `unsafe.Pointer` type (*“Support for channels, maps and unsafe.Pointers”* [7]). As shown in RQ2, developers frequently rely on `unsafe` for optimizations and we found issues open in 5 projects proposing a refactor to more efficient code that uses `unsafe`.

All bug issues are exclusively caused by the use and misuse of `unsafe`. The main takeaway from this analysis is that projects that do not break type-safety and depend on type-safe third-party packages are free of encountering the issues we discussed in this analysis, such as crashing errors caused by bad pointers, have portability and compatibility issues or have their code restricted to deploy in different environments.

Summary of RQ3: We found suspicious cases of invalid pointer conversions in 14 out of 211 projects. More importantly, projects that use `unsafe` face several exclusive issues, from having their deployment restricted (20 projects), experiencing runtime errors that are hard to reproduce (16 projects), introducing bugs due to misuse of `unsafe` (12 projects).

7 RECOMMENDATIONS

Our results show that `unsafe` usage is widespread, motivated by low-level software integration and performance optimization. However, it puts projects at the risk of several issues. In this section, we draw a series of recommendations derived from our results that could help make Go language more safe.

Inclusion of more powerful static analyzers. Currently, the tool Go Vet focus on identifying invalid pointer conversions, one of the main issues with using `unsafe`. However, researchers and practitioners can develop analyzers that attempt to find other invalid cases of `unsafe` usage, such as using `unsafe.Pointer` to dereference a nil pointer or access a memory address beyond the allocated memory space. For instance, statically identifying null pointers is a problem well investigated in languages like C and C++ [23], [36], and similar approaches could be employed in Go to identify `unsafe` nil pointer conversion. In fact, the newest Go release (1.14) already provides a more robust set of checks, by embedding a compile option that instrument the code to capture violation of safety rules.

Furthermore, our study shows that the majority of `unsafe` usages are concentrated on a handful of patterns. The catalog of usages yielded by our study can be employed in static analysis tools to identify miuses on widely-used patterns. For instance, the slice and bytes-string conversion should draw special pattern to static analyzers, as developers often implement such conversions without guarding for invalid pointer conversion, causing their programs to crash and behave erroneously.

Improve documentation on frequent `unsafe` usages. Our investigation shows that the conversion of string to bytes (and vice-versa) is a very common optimization method using `unsafe`, as it provides substantial performance improvement when reading/serializing strings. However, the official documentation of `unsafe` only briefly touches the fundamentals of the issues developers encounter when writing this code. This is further corroborated by discussions in mailing lists related to Go on how to properly convert string to bytes [27] and proposals to clarify the usage of `unsafe` when performing syscalls [22]. While Go language maintainers have expressed that including such conversions in the standard API is not desired [44], due to the issues of breaking string immutability, an official statement on how to proper convert between string and bytes could auxiliate developers at writing a correct code conversion and finding the ill-implemented variants.

Inclusion of Generics. We found some usages of `unsafe` to be primarily motivated by the type flexibility a support

to generics would provide. Without generics, generic form functions (e.g., customized ways of sorting a slice) are accomplished with interfaces, reflection or code generation [11]. The standard marshaling package relies on runtime reflection to encode and decode json to Go structs, which can negatively impact the performance of object serialization. The inclusion of generics would allow for more efficient encoders/decoders without relying on `unsafe`, as the compiler could generate code automatically based on type specified by developers in their generic functions [8].

Planning `unsafe` breaking changes. There exists several proposals that if implemented in Go 2.0 may impact programs that currently use `unsafe` [35], [58], introducing breaking changes in the language. Our study shows that a quarter of the most popular Go projects use `unsafe` in their code, and given their notoriety, it is expected that such projects are used by a large share of the Go community. We build a dependency graph using the Go List command, considering only the project in our dataset (2,438 most popular Go projects), and found that 40% of the projects either use `unsafe` directly or depend directly on a project that uses `unsafe`.

Fortunately, our study also shows that the majority of the `unsafe` usage in popular Go projects is well-localized in code, most projects concentrate their `unsafe` in at most 2 files. This indicates that the cost for updating the `unsafe` usages to comply to possible breaking changes in Go 2.0 version should be manageable for most projects. Still, language designers could mitigate the cost for adopting the new language version by communicating the adopted proposals in advance to the community, or even better, by employing a tool for migrating some of the `unsafe` usages to the newest package version within the Go Fix command [1]. The Go Fix is a tool created with the sole purpose of migrating old APIs to new ones in the case of breaking changes in the language, and can be used to update `unsafe` usages from valid Go 1.x code to Go 2.x.

8 RELATED WORK

There exists a plethora of work that investigate how developers use certain features of the programming language. In this section, we discuss the work that is most related to our study. We grouped prior work into work related to `unsafe` features in programming language and studies on mining repositories to assess language features.

8.1 Unsafe Language Features

Some prior studies investigate how developers use low-level code features in different programming languages. Nagappan et al. [50] investigate how developers use the `goto` statements in a representative sample of C programming files. Motivated by the harmful stigma of `goto` statements, the authors qualitatively investigate how such command is used in modern C code. The investigation showed that developers nowadays limit themselves to use `goto` only in very specific circumstances, such as error handling and cleaning up resources, and that it does not appear to be harmful in practice.

Some other studies focus on investigating the usage of dynamic features on programming languages. Mastrangelo et al. [46] investigated how and when developers use casting on thousands of Java projects. The results show that casting is widely-used by Java developers and that half the casts are not guarded locally to ensure against runtime errors. Similarly to our study, this investigation also cataloged a common set of use-case patterns that can help language maintainers and tool developers to better accommodate the most common usages of dynamic casting. Callau et al. [15] investigated how dynamic and reflective features are employed in SmallTalk, by surveying a thousand SmallTalk projects. While SmallTalk is a dynamically-typed language, the authors reported that dynamic features are not widely used by SmallTalk developers. In fact, the two most pervasively used features are the ones other static languages implement, indicating a more conservative approach with unsafe features than the ones we observed in popular Go projects.

The work most closely related to ours in the investigation of unsafe APIs in Java by Mastrangelo et al. [47]. In their work, the authors investigated how developers use the `sun.misc.Unsafe`, a class that exposes low-level and unsafe features, violating Java safety guarantees. Differently from the `unsafe` package in Go, the `sun.misc.Unsafe` API gives access to more than type-safety violations, for instance, developers can violate method contracts by throwing a checked exception undeclared by a method. Their investigation found that only a small share (1%) of software artifacts use the `unsafe` API in their code directly. In contrast, we found that a 24% of the most popular Go projects rely on `unsafe` to some extent, showing that break type-safety is way more common in Go than in Java.

8.2 Mining Repositories of Usage of Language Features

Several studies empirically investigate how developers use different language features through mining software repositories. Mazinianian et al. [48] mined software repositories to investigate how Java developers use Lambda in their programs. Similarly, Costa et al. [19] profiled how developers select and tune their data structures in Java programs, showing that developers only rarely tune their data structures and prefer the standard implementations, despite having better performance variants available. Krikava et al. [42] investigate the usage of implicits - a language feature that allows developers to reduce boilerplate code - in Scala programs. The results showed a pervasive use of the feature in Scala projects hosted in GitHub. Guilherme and Weiyl [20] performed an empirical study to examine the prevalence of exception-handling anti-patterns in Java and C# projects. Their findings showed that all studied projects contain exception handling anti-patterns in their source code. Wang et al. [59] studied the evolution regular expressions over time from the different aspects. Their results showed that the use of regular expression is stable in the development history of the studied GitHub projects.

The aforementioned work mines software repositories to quantitatively assess the level of adoption of language features and libraries from developers. Our study shares part

of this mining repository methodology, such as selecting studied projects, parsing the code to get quantitative results and manually analyze the usage patterns for a qualitative assessment.

9 THREATS TO VALIDITY

This section describes the threats to the validities of our study.

9.1 Internal validity

Threats to internal validity are related to experimenter bias and errors. To identify `unsafe` usages, we build a customized parser which could miss or introduce false positives in our analysis. To mitigate this threats, we first took special care to identify all possible ways `unsafe` can be used in Go code, by inspecting import statements for `unsafe` which developers are required to do to use the package. Second, during our manual examination of the detected cases of `unsafe` to catalog the usage patterns, we only found a single case (1/270) of false positive, which give us a confident in the accuracy of the build customized parser. Developers had defined a custom object named “unsafe” and called a method from the object, filtered in by our parser. This represents a single case in 270 cases, and since developers are discouraged to name packages with “unsafe”, we believe our results hold. We also use the tool Go Vet to identify suspicious cases of invalid pointer conversions. While we have manually assessed that 2 out of 16 cases were false-positives, the tool could miss real cases due to the complexity of statically analyzing the code. Hence, our results with Go Vet should be interpreted as a lower bound of invalid pointer conversions.

Furthermore, we conduct two major manual analyses in our study, to investigate the most used usage patterns and to classify `unsafe`-related issues. In the classification of the `unsafe` usage patterns, we include a single random file per project, hence, our analysis is based on the usage of a single randomly selected file from a sample of representative projects. While this ensured we performed a cross-project analysis, mitigating the risks or selecting multiple use-cases from projects that have hundreds of `unsafe` usages, this result should be interpreted as an initial assessment of `unsafe` usage patterns.

Regarding the analysis of `unsafe`-related issues, it is important to note that the categories of issues obtained in this analysis are not exhaustive or necessarily representative. We only investigate issues that have the keyword “unsafe” in its title and hence are bound to miss many `unsafe`-related issues. Hence, our analysis is a lower bound of possible `unsafe`-related issues, providing qualitative support to the risks of using `unsafe` entails to software projects.

9.2 External validity

Threats to external validity are related to the generalizability of our findings. Our investigation focused on the most popular Go projects. Go has established itself as a programming language for high-performance infrastructure projects and the projects in our dataset reflect that (e.g., there is a relative low number of front-end projects). The

`unsafe` package is used for low-level implementation and optimization, hence its prevalence is expected to decrease on less popular projects or on a more diverse set that may not have the pressure for high-performance code. We argue that, while not fully generalizable to all Go software projects, our dataset contains the most influential projects of the current Go landscape. Also, in our analysis, we examine open-source Go projects that are hosted on GitHub. Thus, our results may not be generalized to property projects.

10 CONCLUSION

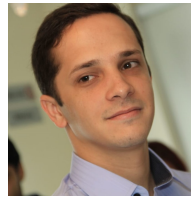
In this paper, we present the first study on the usage of `unsafe` in Go programs. We conduct a mix-method analysis of the prevalence of `unsafe` in popular open-source Go projects, investigate why developers break type-safety, and evaluate some of the real risks projects that use `unsafe` are subjected to. Our results have shown that the use of `unsafe` is prevalent, with one in four Go projects breaking type-safety primarily for communicating with programs outside of Go language (C code and Operating Systems), and optimizing type casting functions. Developers have made the conscious effort of keeping `unsafe` restricted to a selected set of packages in their project, but report a series of exclusive issues related with the use of the package. Their projects may have the deployment restricted by shared cloud environments such as the Google App Engine, developers report crashing errors and wrong results due to invalid pointer conversion, and in some cases the program produces the wrong behavior due to non-deterministic causes.

Our study can be used as empirical evidence on the state of usage of `unsafe` in Go and help motivate further tools and language developments that could make Go safer. We suggest that special attention should be given towards creating tools that identify further bad practices related with the package, similarly to how Go 1.14 introduces more robust checks for invalid pointer conversions, as developers seem to struggle in writing valid `unsafe` code. Language maintainers can also mitigate the encountered issues by documenting official `unsafe` snippets to be used by the community, in particular on type casting of string and bytes, widely used as a source of optimization. Furthermore, the level of adoption of `unsafe` should also be taken into consideration when planning future versions of the language, as the package as risky as it is, seems to be integral to the Go community.

REFERENCES

- [1] fix - the go programming language. <https://golang.org/cmd/fix/>. [Online; accessed on 01/03/2020].
- [2] Go 1 and the future of go programs - the go programming language. <https://golang.org/doc/go1compat>. [Online; accessed on 10/11/2019].
- [3] The go project - the go programming language. <https://golang.org/project/>. [Online; accessed on 03/02/2020].
- [4] pointers - what are the possible consequences of using `unsafe` conversion from `[]byte` to string in go? - stack overflow. <https://stackoverflow.com/questions/33952378/what-are-the-possible-consequences-of-using-unsafe-conversion-from-by> [Online; accessed on 18/12/2019].
- [5] vet - the go programming language. <https://golang.org/cmd/vet/>. [Online; accessed on 10/11/2019].
- [6] Stack overflow developer survey 2019. <https://insights.stackoverflow.com/survey/2019>, 2019. [Online; accessed on 10/11/2019].
- [7] A. Arzilli. Support for channels, maps and `unsafe.pointers` by `aarzilli` pull request #286 go-delve/delve. <https://github.com/go-delve/delve/pull/286>, Nov 2015. [Online; accessed on 01/03/2020].
- [8] T. G. Authors. Go2genericsfeedback. <https://github.com/golang/go/wiki/Go2GenericsFeedback>. [Online; accessed on 03/04/2020].
- [9] T. G. Authors. tools/unsafepttr.go golang/tools. <https://github.com/golang/tools/blob/master/go/analysis/passes/unsafepttr/unsafepttr.go>, Oct 2019. [Online; accessed on 03/04/2020].
- [10] W. Bitter. Invalid use of `unsafe.pointer`. <https://github.com/go-gl/gl/issues/18>, May 2015. [Online; accessed on 03/04/2020].
- [11] V. Blanchon. Go: Is the encoding/json package really slow? - a journey with go - medium. <https://medium.com/a-journey-with-go/go-is-the-encoding-json-package-really-slow-62b64d54b148>, May 2019. [Online; accessed on 01/03/2020].
- [12] V. Blanchon. Go: What is the `unsafe` package? - a journey with go. <https://medium.com/a-journey-with-go/go-what-is-the-unsafe-package-d2443da36350>, Jun 2019. [Online; accessed on 30/05/2020].
- [13] H. Borges, A. Hora, and M. T. Valente. Predicting the popularity of github repositories. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, New York, NY, USA, 2016. Association for Computing Machinery.
- [14] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.
- [15] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: The case of smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR 11, page 2332, New York, NY, USA, 2011. Association for Computing Machinery.
- [16] I. Chekrygin. Prometheus crashes and hangs on fatal error: found bad pointer in go heap (incorrect use of `unsafe` or `cgo`?). <https://github.com/prometheus/prometheus/issues/2263>, Dec 2016. [Online; accessed on 01/03/2020].
- [17] G. Cloud. App engine. <https://cloud.google.com/appengine>. [Online; accessed on 01/03/2020].
- [18] CockroachDB. Cockroachdb - the open source, cloud-native sql database. <https://github.com/cockroachdb/cockroach>. [Online; accessed on 01/03/2020].
- [19] D. Costa, A. Andrzejak, J. Seboek, and D. Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE 17, page 389400, New York, NY, USA, 2017. Association for Computing Machinery.
- [20] G. B. de Pádua and W. Shang. Studying the prevalence of exception handling anti-patterns. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC 17, page 328331. IEEE Press, 2017.
- [21] M. Dempsy. Guarantees for package `unsafe`. <https://bit.ly/39WPORD>, Aug 2016. [Online; accessed on 01/03/2020].
- [22] M. Dempsy. proposal: `unsafe`: clarify `unsafe.pointer` rules for package `syscall`. <https://github.com/golang/go/issues/34684>, Oct 2019. [Online; accessed on 01/03/2020].
- [23] F. C. Eigler. Mudflap: Pointer use checking for c/c++. In *GCC Developers Summit*, page 57. Citeseer, 2003.
- [24] fiatjaf. Strange error happening in just one function of a go library after compiled. <https://github.com/gopherjs/gopherjs/issues/642>, May 2017. [Online; accessed on 03/04/2020].
- [25] S. Fincher and J. Tenenbergs. Making sense of card sorting data. *Expert Systems*, 22(3):89–93, 2005.
- [26] T. G. Foundation. The gtk project - a free and open-source cross-platform widget toolkit. <https://www.gtk.org/>. [Online; accessed on 03/02/2020].
- [27] Francis. Clarification on `unsafe` conversion between string and `[]byte`. <https://bit.ly/2VHHoJN>, Sep 2019. [Online; accessed on 01/03/2020].
- [28] S. Frei. Flaky “`unsafe` pointer conversion”-panics with `-race` and `go1.14rc1`. <https://github.com/spaolacci/murmur3/issues/29>, Feb 2020. [Online; accessed on 01/03/2020].
- [29] Golang. atomic - the go programming language. <https://golang.org/pkg/sync/atomic/>. [Online; accessed on 18/12/2019].

- [30] Golang. cgo - the go programming language. <https://golang.org/cmd/cgo/>. [Online; accessed on 18/12/2019].
- [31] Golang. reflect - the go programming language. <https://golang.org/pkg/reflect/>. [Online; accessed on 18/12/2019].
- [32] Golang. syscall - the go programming language. <https://golang.org/pkg/syscall/>. [Online; accessed on 18/12/2019].
- [33] Golang. unsafe - the go programming language. <https://golang.org/pkg/unsafe/>. [Online; accessed on 18/12/2019].
- [34] golang. golang: Go bindings for gtk3. <https://github.com/golang/gtk3>. [Online; accessed on 03/02/2020].
- [35] R. Griesemer. proposal: spec: disallow t-uintptr conversion for type t unsafe.pointer. <https://github.com/golang/go/issues/20171>. [Online; accessed on 01/08/2019].
- [36] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):1319, Sept. 2005.
- [37] J. K. III. Removed usage of package "unsafe" to allow google app engine compatibility. <https://github.com/tidwall/gjson/pull/69>, Apr 2018. [Online; accessed on 01/03/2020].
- [38] J. K. III. runtime error: unsafe pointer arithmetic (go 1.14 checkptr). <https://github.com/yuin/gopher-lua/issues/254>, 2018. [Online; accessed on 28/05/2020].
- [39] G. Inc. The state of the octoverse — the state of the octoverse celebrates a year of building across teams, time zones, and millions of merged pull requests. <https://octoverse.github.com/>. [Online; accessed on 01/08/2019].
- [40] A. Issa. Google app engine failed parsing input: parser: bad import "unsafe". <https://github.com/hashicorp/go-memdb/issues/44>, Dec 2017. [Online; accessed on 01/03/2020].
- [41] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, Oct 2016.
- [42] F. Krikava, H. Miller, and J. Vitek. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proc. ACM Program. Lang.*, 3(OOPSLA):163:1–163:28, Oct. 2019.
- [43] D. Lemire. Keep header.data alive during unsafe operations. <https://github.com/RoaringBitmap/roaring/pull/226>, Aug 2019. [Online; accessed on 03/04/2020].
- [44] T. Liron. Feature: provide no-copy conversion from []byte to string issue. <https://github.com/golang/go/issues/25484>, May 2018. [Online; accessed on 01/03/2020].
- [45] C. Mastrangelo. proposal: sync/atomic: add swap, compare-and-swap. <https://github.com/golang/go/issues/26728>, July 2018. [Online; accessed on 03/04/2020].
- [46] L. Mastrangelo, M. Hauswirth, and N. Nystrom. Casting about in the dark: An empirical study of cast operations in java programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):158:1–158:31, Oct. 2019.
- [47] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM.
- [48] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017.
- [49] M. McHugh. Interrater reliability: The kappa statistic. *Biochemia medica : asopis Hrvatskoga drutva medicinskih biokemiara / HDMB*, 22:276–82, 10 2012.
- [50] M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan. An empirical study of goto in c code from github repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 404–414, New York, NY, USA, 2015. ACM.
- [51] C. Project. Cilium. <https://cilium.io/>. [Online; accessed on 01/03/2020].
- [52] V. Romanov. Unsafe use of unsafe that leads to data corruption. <https://github.com/francoispqt/gojay/issues/3>, Apr 2018. [Online; accessed on 03/04/2020].
- [53] T. Sandall. Remove use of unsafe.pointer for string hashing by tsandall. <https://github.com/open-policy-agent/opa/pull/602>, Feb 2018. [Online; accessed on 03/04/2020].
- [54] W. Schulze. No unsafe by awaltersschulze pull request #343 gogo/protobuf. <https://github.com/gogo/protobuf/pull/343>, Oct 2017. [Online; accessed on 01/03/2020].
- [55] B. Siegert. unsafe.sizeof now returns uintptr issue #3 lxn/walk. <https://github.com/lxn/walk/issues/3>, Jun 2011. [Online; accessed on 01/03/2020].
- [56] D. Smirnov. Known issue: unsafe (marshaler and unmarshaler) test failures on big endian architectures. <https://github.com/gogo/protobuf/issues/195>, Aug 2016. [Online; accessed on 01/03/2020].
- [57] C. Stockton. [security] incorrect unsafe usage potentially exposes prior request parameters. <https://github.com/nucio/nucio/issues/277>, Oct 2017. [Online; accessed on 03/04/2020].
- [58] I. L. Taylor. proposal: Go 2: only give special properties to unsafe.pointer if package imports unsafe. <https://github.com/golang/go/issues/26070>. [Online; accessed on 01/08/2019].
- [59] P. Wang, G. R. Bai, and K. T. Stolee. Exploring regular expression evolution. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–513. IEEE, 2019.



Diego Elias Costa is a postdoctoral researcher at the DAS Lab, in the Department of Computer Science and Software Engineering at Concordia University. His research interests cover a wide range of software engineering and performance engineering related topics, including mining software repositories, empirical software engineering, performance testing, memory-leak detection, and adaptive data structures. You can find more about him at <http://das.encs.concordia.ca/members/diego-costa/>.



Suhaib Mujahid is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University. He received his masters in Software Engineering from Concordia University (Canada) in 2017, where his work focused on detection and mitigation of permission-related issues facing wearable app developers. He did his Bachelors in Information Systems at Palestine Polytechnic University. His research interests include wearable applications, software quality assurance, mining software repositories and empirical software engineering. You can find more about him at <http://users.encs.concordia.ca/smujaahi>.



Rabe Abdalkareem received his PhD in Computer Science and Software Engineering from Concordia University, Montreal, Canada. His research investigates how the adoption of crowd-sourced knowledge affects software development and maintenance. Abdalkareem received his masters in applied computer science from Concordia University. His work has been published at premier venues such as FSE, ICSME and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. Contact him at rab_abdu@encs.concordia.ca; <http://users.encs.concordia.ca/rababdu>



Emad Shihab is an associate professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihab's research interests are in Software Quality Assurance, Mining Software Repositories, Tech-

nical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and

ACM. More information can be found at <http://das.encs.concordia.ca>.