UnGoML: Automated Classification of `unsafe` Usages in Go

Anna-Katharina Wickert
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
wickert@cs.tu-darmstadt.de

Clemens Damke
Institute of Informatics
University of Munich
Munich, Germany
clemens.damke@ifi.lmu.de

Lars Baumgärtner
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
baumgaertner@cs.tu-darmstadt.de

Eyke Hüllermeier
Munich Center for Machine Learning
Institute of Informatics, University of Munich
Munich, Germany
eyke@lmu.de

Mira Mezini
Hessian Center for Artificial Intelligence (hessian.AI)
National Research Center for Applied Cybersecurity ATHENE
Software Technology Group, Technische Universität Darmstadt
Darmstadt, Germany
mezini@cs.tu-darmstadt.de

***Abstract*—The Go programming language offers strong protection from memory corruption. As an escape hatch of these protections, it provides the `unsafe` package. Previous studies identified that this `unsafe` package is frequently used in real-world code for several purposes, e.g., serialization or casting types. Due to the variety of these reasons, it may be possible to refactor specific usages to avoid potential vulnerabilities. However, the classification of `unsafe` usages is challenging and requires the context of the call and the program's structure. In this paper, we present the first automated classifier for `unsafe` usages in Go, UnGoML, to identify *what* is done with the `unsafe` package and *why* it is used. For UnGoML, we built four custom deep learning classifiers trained on a manually labeled data set. We represent Go code as enriched control-flow graphs (CFGs) and solve the label prediction task with one single-vertex and three context-aware classifiers. All three context-aware classifiers achieve a top-1 accuracy of more than 86% for both dimensions, `WHAT` and `WHY`. Furthermore, in a set-valued conformal prediction setting, we achieve accuracies of more than 93% with mean label set sizes of 2 for both dimensions. Thus, UnGoML can be used to efficiently filter `unsafe` usages for use cases such as refactoring or a security audit.**
UnGoML: https://github.com/stg-tud/ungoml
Artifact: https://dx.doi.org/10.6084/m9.figshare.22293052

***Index Terms*—graph neural networks, Go, unsafe package, classification, API-misuse**

I. Introduction

In November 2022, NSA released guidance on how to avoid memory vulnerabilities, such as buffer overflows, as these still occur very frequently in code [1]. One of the recommendations is to use modern programming languages, such as Java, Rust, and Go, with automatic memory management to avoid these vulnerabilities. But they also provide escape hatches, such as the `unsafe` API, for several purposes. For instance, empirical studies on the usage of Go's `unsafe` API in-the-wild [2, 3] show that `unsafe` is used for calling C libraries, making system calls, serialization, performance optimizations, or for modeling missing language features such as generics[1].

Such escape hatches may, however, reintroduce security risks similar to those in memory unsafe languages [2, 4, 5, 6]. Hence, their usage requires special attention by software quality teams and developers and should be used rarely. However, in practice the usage is wide-spread across Go projects [2] and occur more frequently than expected by Go experts [3]. Lauinger et al. [2] analyzed 343 top-rated GitHub projects for two potentially exploitable usage patterns and identified 60 usages of this pattern. The following report revealed a fix and response rate of over 70 % [2]. Further, the Go community actively engages and educates about the risks, e.g., in blog posts or talks at their main conference[2].

To mitigate the risk, auditing the usages and refactoring to safer alternatives should be considered. The `unsafe` API as well as the language evolves, and one may want to perform a large-scale refactoring to introduce the safer alternatives into the code base. For example, Generics were recently introduced in Go and can replace the `unsafe` usages that served to model generics. Also, one might want to check if `unsafe` usages for performing serialization can be replaced with one of the many libraries for efficient serialization. Unfortunately, some `unsafe` usages are unsuitable for large-scale refactoring. Thus, one may want to start an in-depth security audit. A simple search to identify unsafe usages that are flagged as false positives for a static analyzer, reveals that over 370 `unsafe` usages on GitHub are explicitly marked as audited[3].

A prerequisite for such "hardening" actions is efficiently getting an overview of *what* `unsafe` is used for, e.g., to perform pointer arithmetic or to access memory to compare

[1]The latter have been introduced recently, in Go version 1.18, March 2022.
[2]We added details to the README-file of UnGoML: https://github.com/Cortys/unsafe_go_study_results.
[3]See footnote 2.

two addresses, or *why*, e.g., for improving efficiency or for modeling generics. Reasoning manually about the *what* and *why* of `unsafe` usages in large-scale software repositories is cumbersome and time-consuming. At the same time, automated inference of the *what* and *why* of `unsafe` usages is often challenging to be precisely modeled as rules for all `unsafe` usage patterns for traditional static analyzers. Currently, the static analyzers only support a small subset of well-studied security-critical `unsafe` usages [2, 7, 8]. Machine learning methods, in general, and deep learning models, in particular, are typically employed to enable the automated handling of complex problems, for which precise modeling is not feasible. Such methods are used for various software engineering tasks, e.g., for vulnerability detection [9, 10, 11].

The work presented in this paper proposes that modern machine learning classifiers, e.g., graph neural networks (GNNs), are well-suited for the automatic classification of the *what* and *why* of `unsafe` usages – by considering the context of the calls and the structure of the broader program they have the potential to derive meaningful classifications similar to humans. We validate this proposition by designing and implementing UNGOML – a tool support quality assurance teams and auditors to obtain an overview of the tasks solved with `unsafe`. UNGOML builds upon a large set of manually labeled data of `unsafe` usages in real-world code [2] and classifies any `unsafe` usage along two dimensions, namely *what is done* and *why* `unsafe` is used.

We investigated three different GNN architectures, more specifically, DeepSets [12], graph isomorphism networks (GINs) [13], and the higher-order 2-WL-GNN architecture [14]. A GNN architecture seems natural given that code is typically represented in graphs, and recent work on vulnerability classification has shown that GNNs can improve the accuracy [15, 16, 17] over token-based approaches. The different GNN models use the control-flow and edge information differently, e.g., the DeepSets classifier [12] ignores the control-flow structure and variable usages between the input vertices. This enables us to assess the importance of structural information.

Finally, to effectively support developers and auditors even for ambiguous predictions, we investigate conformal prediction [18], a framework for reliable prediction that comes with statistical guarantees. This technique allows for predicting a set of candidate labels covering the true classification for each usage with a prespecified probability, e.g., 95%. Thus, we can present a set of varying sizes instead of a fixed one. Such sets reflect the fact that sometimes even human annotators have difficulties agreeing on one specific label [3]. Further, we assume that conformal set prediction can improve the usability for an end user over the top-1 or top-3 accuracy. The underlying assumption is that the conformal set prediction finds the balance between achieving high accuracy while keeping the set of potential labels as small as possible. To the best of our knowledge, this work is the first to explore conformal set prediction [18] for software engineering tasks.

Our results positively validate our proposition that machine learning models are well-suited to classify automatically the what and why of `unsafe` usages. For all three context-aware models, we achieve a combined (`WHAT` and `WHY`) top-1 accuracy of nearly 80% and a top-3 accuracy of over 91%. For the single-vertex model multilayer perceptron (MLP), the top-1 accuracy drops to 74%. This indicates that the context is relevant for classifying `unsafe` usages. Further, the average sizes of the sets predicted by the conformal set prediction are around 2 elements with a combined accuracy of over 93%. Thus, conformal set predictions are suitable for `unsafe` classification to improve the accuracy at the cost of a flexible but on average small set of labels.

We integrated our prediction models along with GOGEIGER [2], into a tool to identify `unsafe` usages along with the classification of any usages of `unsafe` in a given Go project as the basis for further refactoring or security auditing. In summary, we make the following contributions:

- A formalization of the problem of inferring programmers intentions when using `unsafe` (`WHAT` and `WHY`) as a classification problem.
- A comparison of relational and non-relational machine learning models, such as GNNs and DeepSets, to understand the impact of the call graphs on the prediction.
- A discussion of the important features needed to classify the what and why of `unsafe` usages.
- UNGOML, the first classification tool that predicts each `unsafe` usage within Go projects to understand how and why `unsafe` is used to support developers and auditors in effectively filtering and judging `unsafe` usages.
- Initial evidence that conformal set prediction seems valuable and worth exploring for software engineering tasks that leverage classifiers.

II. UNSAFE USAGE PATTERNS

In this section, we introduce Go's `unsafe` package along with results [2, 3] on its usages (Subsection II-A), the labeled data set used to train our model (Subsection II-B), and one example of an `unsafe` usage (Subsection II-C).

A. Unsafe In Go

Type Safety and the `unsafe` Package. Go is a statically-typed language. Like in other type-safe languages, e.g., Java or Rust, the `unsafe` package provides a way to enable developers to write low-level code and escape type-safety.

The API consists of five functions and one type [19]. The type `Pointer` represents a pointer type that is more powerful than a "classical" pointer in Go and enables to read and write arbitrary memory. The three functions `Alignof`, `Offsetof`, and `Sizeof` provide information about the memory alignment of Go types. These three functions and one type are discussed in previous studies [2, 3]. The Go 1.17 release (August 2021) added two new functions to the `unsafe` package to supposedly simplify the correct usage of the type `unsafe.Pointer`. As the previous studies did not cover these additions, our discussions of the `unsafe` package focus on the functions and types introduced before Go 1.17.

TABLE I: An overview of different `unsafe` usage labels observed in GitHub projects.

(a) An overview of the possibilities for label `WHAT` (what is done).

Usage	Description	Code Example	C [3]	L [2]
Cast	Implement casts between types	`o = (*int32)(unsafe.Pointer(i))`	●	●
Definition	Declaration `unsafe.Pointer`	`var p unsafe.Pointer`	○	●
Delegate	Pass `unsafe` variable	`needPointer(ptr)`	○	●
Memory-access	Manipulate or reference memory	`d := *((*unsafe.Pointer)(ptr))`	●	●
Pointer-arithmetic	Perform arithmetic change of addresses	`u := uintptr(unsafe.Pointer(&v[0])) & 3`	●	●
Syscall	Use packages for syscalls		●	●
Unused	Dead code or unused parameters	`func A(ptr unsafe.Pointer){}`	○	●

(b) An overview of the possibilities for label `WHY` (underlying purpose of the usage).

Usage	Description	C [3]	L [2]
Atomic Operations	Use `atomic` package	○	●
Avoid Garbage Collection	Prevent free of value	○	●
Efficiency	Improve efficiency of program	◐	●
Foreign Function Interface	Integrate C code	●	●
Generics	Implement generic functionality	○	●
Hide Escape Analysis	Break escape analysis chain	○	●
Memory Layout Control	Manage low-level memory	◐	●
Reflection	Use or re-implement reflection	●	●
Serialization	Implement marshalling and serialization	●	●
Types	Implement Go type systems (std lib)	○	●
Unused	Dead code or unused parameters	○	●

Note, that we combined the different casts, such as *cast-basic* and *cast-bytes* for brevity for `WHAT` and omit code examples for label `WHY` as these are already covered within one of the `WHAT` category examples.
Legend:
C [3] and L [2] presents if this usage pattern was discussed by Costa et al. [3] and Lauinger et al. [2], respectively.
●: covered,
◐: only covered partly,
○: not covered.

By using the `unsafe` package, developers gain more control over the memory at the cost of potentially introducing vulnerabilities in code, e.g., buffer overflows. Besides vulnerabilities, the program may behave differently than expected, e.g., crash. In addition, the usage may render the program to be not portable to different systems as well as not being protected by the Go 1 compatibility guideline [19].

Usage of `unsafe` in Go software. The `unsafe` package is used frequently in popular Go projects on GitHub. Previous empirical studies [3, 2] revealed that 24% to 38% of the projects use `unsafe` within the application code. Furthermore, 91% of the projects use `unsafe` in transitively imported packages [2] with an average depth of 3.08 ± 1.62 [2].

B. Labels for unsafe.Pointer Usages

The manual analyses conducted by Costa et al. [3] and our previous work [2] reveal several usage patterns for `unsafe`. These patterns were observed in a diverse set of applications collected from GitHub and represent usages that occur in-the-wild. We present these patterns along with code examples in Table I. The table is divided into two label categories [2] and includes the usage, a description, a code example, and information if the pattern was discussed in previous work.

The first label category, hereafter `WHAT` (Table Ia), labels what is actually done with the `unsafe` usage. One use case for `unsafe` is to perform casts from arbitrary types to other types, basic types, slices, or `unsafe.Pointer` values. In Table Ia, we grouped them within the label *cast*. As each `unsafe` usage is labeled, it is possible that a usage "only" declares an `unsafe.Pointer` without using it further at this location (*definition*). Similarly, to *definition*, it is possible that a usage "only" passes an `unsafe` variable (*delegate*), e.g., as a parameter. The label *memory-access* groups all `unsafe` usages that manipulate or reference memory. *Pointer-arithmetic* contains `unsafe` usages that perform arithmetic changes of addresses, e.g., advancing an array. For the interaction with low-level operating system primitives, calls to the `syscall` package are necessary, and some functions require `unsafe` parameters to work correctly (*syscall*). Finally, *unused* includes all `unsafe` usages that are dead code or unused parameters.

The second label category, hereafter `WHY` (Table Ib), focuses on the rationale for the usage. The package `atomic` requires `unsafe` pointers for some of their functions. Therefore, developers have to use the `unsafe` package to interact with the library. Go has a garbage collector (GC), and in some cases developers want to prevent that a value is collected by the GC (*avoid garbage collection*) with the help of `unsafe` usages. The label *efficiency* groups usages that aim to improve the time or space of the code. Costa et al. [3] focuses on optimizations due to cast operations. While this holds for the majority of usages classified as *efficiency* by us [2], we included cases such as *memory-access*. The label *foreign function interface* (FFI) marks usages that interact with C code, e.g., by calls that expect unsafe pointers. During the study, generics were not part of the language. Thus, the label *generics* groups `unsafe` usages where developers implement some generics functionality by themselves. The Go compiler has a phase for escape analysis, and in some cases, the developer wants to break the escape analysis chain to improve efficiency [2], which is labeled as *hide escape analysis*. The label *memory layout control* marks usages that aim to manage the memory. The usage pattern by Costa et al. [3] includes examples for getting the memory address, while our patterns [2] also include examples for delegation and definition. The label *reflection*

```
1   // toAddrPointer converts an interface to a pointer that points to the interface data.
2   func toAddrPointer(i *interface{}, isptr bool) pointer {
3           if isptr {
4                   return pointer{p: unsafe.Pointer(uintptr(unsafe.Pointer(i)) + ptrSize)}
5           }
6           ...
7   }
```

(a) A usage of `unsafe` in a frequently used *protobuf* fork [20].

```
8   func makeMapMarshaler(f *reflect.StructField)
    ↪  (sizer, marshaler) {
9     ...
10    vaddr := toAddrPointer(&vi, valIsPtr)
11    ...
12  }
```

(b) One usage of the function with `unsafe` in the project [21].

```
13  // In pointer_reflect.go we use reflect instead of
    ↪  unsafe to implement the same (but slower)
    ↪  interface.
14  type pointer struct {
15          p unsafe.Pointer
16  }
```

(c) An indication that `unsafe` is used to improve performance [20].

Listing 1: An example of an `unsafe` usage that challenges classification. We removed some comments for brevity.

groups usages that use the `reflect` package or implement some reflective functionality. Usages that (un)marshal or (de)serialize are grouped within the label *serialization*. As the labeled data set includes usages of the standard library of Go, some of the `unsafe` usages implement the type system of Go (*types*). Like the `WHAT` label, we group usages that are dead code or unused parameters as *unused*.

C. Example of an `unsafe` usage

Below, we briefly discuss an `unsafe` usage (Listing 1). The function `toAddrPointer` (Listing 1a, Line 2) casts an empty interface, which can be any type, into a pointer to the data of the interface [20]. Line 4 takes the passed interface (`i`) to retrieve an `unsafe.Pointer` that is cast to a `uninptr` and back to a `unsafe.Pointer`. For the conversion back to an `unsafe.Pointer` it is necessary to add the offset (`ptrSize`). The retrieved `unsafe.Pointer` is used to initialize the `pointer` struct in Line 14, which is returned by the method `toAddrPointer`. This function is called (Listing 1b, Line 10) by the function `makeMapMarshaler` (Listing 1b, Line 8) that marshals a map.

For `WHAT`, we classified the usage in Line 4 as *pointer-arithmetic*. In our previous work [2], we decided for this label because pointer arithmetic was necessary to cast back to `unsafe.Pointer`. Another possible label is a *cast* due to the conversion. The `WHY` label for the usage in line 4 is *serialization*. We decided for this label as the caller of the function (Listing 1b, Line 8) marshals a message. Nonetheless, it could be argued that `WHY` for the usage is *efficiency*, because in the global context of the program, `unsafe` is used to improve the efficiency in comparison to the reflection-based implementation as indicated in Line 13.

This example illustrates two possible refactorings to harden or even avoid the `unsafe` usage. First, in Line 4, one can use the function `Add`, which was newly introduced in Go and can be used to wrap the pointer arithmetic. Second, it may be possible to refactor the usage of `unsafe` entirely by evaluating currently existing marshaling libraries. For both cases, it is essential to identify and classify `unsafe` usages.

III. Unsafe Code Classification

This section presents our approach to automatically classify a given `unsafe` usage, thereby answering the two questions "What is happening?" and "For what purpose?". Figure 1 shows a high-level overview of the composition of UnGoML. Section III-A focuses on the representation of an `unsafe` usage, Section III-B then describes how this representation is used to classify the usage.

A. Code Representation

We represent `unsafe` usages as enriched control-flow graphs (CFGs), which encode information about the usages and their surroundings. Thus, our approach follows a recent trend observed for vulnerability detection, that programs are encoded into a CFG variant [10]. We developed our CFG representation by investigating information that is relevant to our problem and can be easily derived. Given `unsafe` usages as pointers to lines of Go code, their graph representation contains the control-flow structure of their surrounding context. Possible contexts of an usage are either the body of the function, the type declaration, or the global variable definition where the usage occurs. In cases where the `unsafe` usage context is a type declaration or a global variable definition, there is no control-flow structure, and the context is represented as a single statement vertex of type `declaration`.

There are two types of vertices in the CFG representation: *Statement* vertices and *variable* vertices. *Statement vertices* correspond to Go statements; they are connected via `flow` and `alt-flow` edges, which indicate possible execution paths – the latter represent the control-flow from a branching statement to its successor if the branch condition is not satisfied; `flow` edges represent all other control-flow relations. *Variable vertices* correspond to the *named* memory locations referenced by statement vertices; this includes stack and heap variables, as well as struct fields and function pointers. Edges from statement vertices to variable vertices represent different types of memory accesses, namely `decl` edges for variable declarations, `use`/`dir-use` for reads, `update`/`assign` for writes, and `call` for function pointer calls.

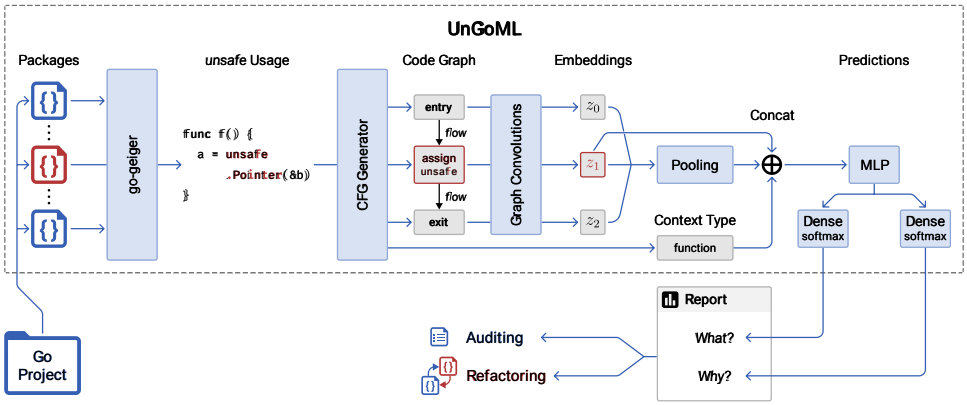

Fig. 1: High-Level Design of UnGoML.

There are two types of read access edges to distinguish between tail and non-tail positions in pointer dereference chains of the form `x = s.f1.f2.f3`; the three non-tail dereferences would be represented by `use` edges, while the final dereference of the field `f3` would be represented by a `dir-use` edge. Analogous, there are two types of write edges to distinguish between direct ("real") writes to memory and indirect variable modifications; a statement of the form `s.f1.f2.f3 = x` would have three `update` edges to the non-tail dereferences and one `assign` edge to the field `f3`.

Lastly, there are `contains` edges from a variable vertex v_a to a variable vertex v_b. They indicate the existence of a pointer to the memory location of v_b at v_a; for the previous example `s.f1.f2.f3`, there would be three `contains` edges in our graph: `s` → `f1`, `f1` → `f2` and `f2` → `f3`.

1) Mapping Go Code to Vertices: Most CFG vertices directly correspond to a single Go statement or variable/field. However, there are two exceptions to this direct correspondence. First, we add a `flow` edge between the pseudo-statement vertex `entry` and the first Go statement and one for all terminal statements, such as `return`, to `exit`. Further, we model function parameters, return values, and receivers as special variable vertices that are declared by the `entry` vertex. `return` statements are treated as assignments to the return variables, with `assign` edges being added between both. Second, we split branch statements that combine a branch condition expression with another statement, as in `for i := 0; i < 6; i++ {...}`, into separate statement vertices. Figure 2 illustrates the CFG vertices that are generated for a simple function.

2) Label-based Vertex Representation: We encode the individual statements and variables by assigning labels to each vertex. Statement vertices can have the following labels:

Statement Types: Each statement vertex has exactly one label representing its type, e.g., `if`, `switch`, `for`, `return`, `assign`, or `declare`.

Data Types: The data types that are instantiated in a statement via `make`, `new`, literal expressions, or casts are represented using distinct labels. For composite types, we also include additional labels for encoding the contained basic types (e.g., `bool` or `float32`) and the used composition structures (e.g., `Struct` or `Slice`). The instantiation of the type `map[string]**[]int` inside a statement would be represented by six labels: `Map`, `string`, `Pointer`, `Slice`, `int`, and one label for the complete composite type.

Operators: Go operators occurring in a statement are represented by corresponding operator labels, e.g., `binary/+`, `binary/==`, `unary/&`, or `unary/-`.

Functions: Both the built-in Go functions (e.g., `len` or `append`) and all other regular package functions (e.g., `fmt.Errorf` or `golang.org/x/sys/unix.Syscall`) called within a statement are represented by distinct labels.

Packages: In addition to the function labels, the origin packages of all called functions in a statement are added as labels, e.g., `fmt` or `golang.org/x/sys/unix`.

Self-references: If a statement in the body of a function f contains a recursive call to f, the `selfref/function` label is added to that statement. If a statement occurring in a Go module m and a package p contains a call to any function from m and/or p, the `selfref/module` and/or `selfref/package` labels are added. Lastly, recursive declaration statements, e.g., recursive structs, are labeled with the `selfref/type` label.

Note that this label-based representation does not preserve the syntactic structure within statements; both statements `x=f(a+b)*g(f(c))` and `x=g(a*f(b)+c)` will be represented by the same set of vertex labels. We also experimented with a more fine-grained representation that encodes the abstract syntax trees (ASTs) of statements. However, this representation did not improve the classification accuracy, while increasing graph sizes and therefore slowing down model training; hence, we chose to discard the AST structure of statements.

Variable vertices are labeled using the following categories:

Variable Types: We model function parameters, results, and receivers as variable nodes. To distinguish those special variables from regular Go variables, we add a `param`, `result`, or `receiver` label to them.

Variable Names: The name of a variable is added as an additional label to each variable vertex, such as `i` for iterators

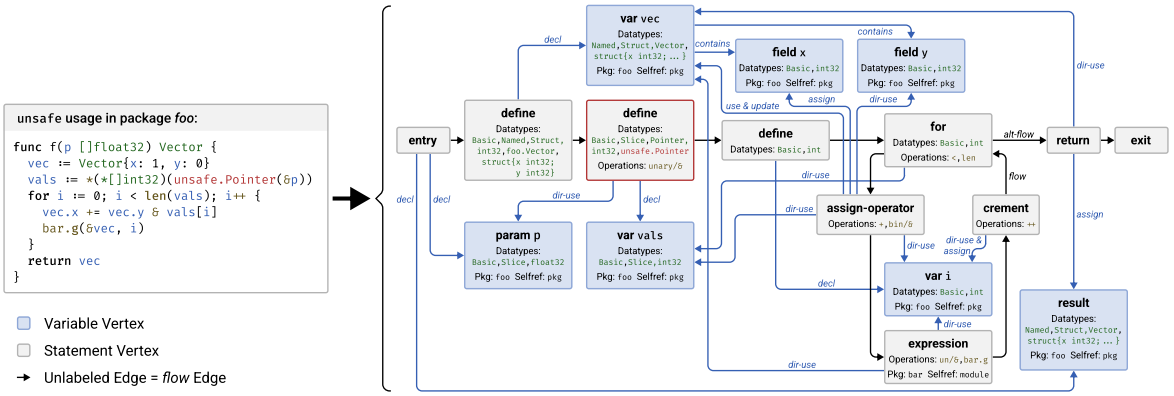

Fig. 2: An exemplary code snippet containing `unsafe` and its corresponding enriched CFG representation. We highlight the "enriched" part of our CFG with the blue nodes and edges, and the `unsafe` usage with red.

or `err` for errors.

Datatypes: We represent the data type of each variable by a set of labels analogously as for statement vertices.

Packages: For each variable, we add the name of the package in which the variable is defined as an additional label.

Self-references: If a variable is defined in the same module and/or package as the context of the considered `unsafe` usage, a `selfref/module` and/or `selfref/package` is added. If the context of an `unsafe` usage is a global variable definition, the variable vertex corresponding to the defined global variable gets the `selfref/variable` label. The CFG in Fig. 2 illustrates how different label types are used.

3) Mapping Vertex Labels to Feature Vectors: One common assumption of machine learning (ML) algorithms is that their input is represented as one or multiple numerical feature vectors of fixed dimensionality. This is a common requirement that also holds for other machine learning models besides GNNs, such as CodeBERT [22]. The labeling scheme described in the last section does not fulfill this assumption; the individual labels are not numerical nor is their number fixed, since there is a potentially infinite number of data types, functions, packages, and variable names.

We address this issue by restricting the set of allowed labels to the most common ones in the training data. More precisely, for each of the label categories the top-k most frequent labels within that category are selected. For finite label categories, k is chosen such that all possible labels within those categories are selected. For the four infinite label categories Datatypes, Functions, Packages and Variable Names, we chose a fixed cutoff of $k = 127$ as we did not observe any improvements by using a larger cutoff in our experiments. If a vertex has an uncommon label that is not part of the per-category top-k selection, that label is replaced by a fallback '`other`' label for its category. For example, variable names such as `err` for errors and `i` for iterators are encoded, while uncommon variable names are grouped within the label '`other`'. This approach reduces the number of considered labels for each infinite category to $k+1 = 128$. Combining the labels of all categories, we obtain a total of $n := 4 \cdot (k+1) + n_{finite} = 594$ possible vertex labels $\mathcal{L} = \{l_i\}_{i=1}^{n}$, where n_{finite} is the number of labels in the finite label categories.

Through this reduction of the permitted labels, the label set $L_v \subseteq \mathcal{L}$ of a vertex v can be encoded as a binary feature vector $x_v \in \{0,1\}^n$, with $x_v[i] := \mathbb{1}[l_i \in L_v]$. Using this encoding strategy, an enriched CFG becomes a directed multigraph with binary vertex feature vectors and nine types of edges.

B. Model Architecture

Our `unsafe` usage classification approach is based on a family of ML models called graph neural networks (GNNs). Over the recent years, GNNs have been successfully applied to various graph learning tasks, including graph classification, and are becoming the standard for software engineering classification tasks such as vulnerability detection [15, 10, 17, 16]. GNN-based graph classifiers are a family of differentiable models, which take graphs with vertex, and depending on the GNN variant, edge feature vectors as input and output a vector encoding the predicted class probabilities. Below, we briefly introduce GNNs and subsequently present how we use them to solve the `unsafe` usage classification problem.

1) Introduction to GNNs: A GNN for graph-level prediction tasks, such as graph classification, typically consists of a sequence of so-called *graph convolution* layers, followed by a *graph pooling* layer. Generally speaking, a graph convolution operator takes a set of vertex feature vectors $\{x_i \in \mathbb{R}^d\}_{i=1}^{n}$ as input[4] and aggregates the feature vector x_i of each vertex v_i with the feature vectors of other vertices v_j that are related to v_i by some structural characteristic in the graph. The result of each convolution is a set of convolved feature vectors $\{z_i \in \mathbb{R}^{d'}\}_{i=1}^{n}$. After applying one or more graph convolutions to the feature vectors, a final graph-level vector representation $z_G \in \mathbb{R}^{d'}$ is obtained by combining the convolved vectors $\{z_i \in \mathbb{R}^{d'}\}_{i=1}^{n}$ via a pooling layer.

The simplest possible graph convolution ignores all structural information and treats a graph as a set of vertices.

[4]There are convolution approaches that also consider edge feature vectors. For simplicity, they will not be covered here.

It aggregates each vertex feature x_i only with itself; this convolution is described by $z_i = f(x_i)$, $f : \mathbb{R}^d \to \mathbb{R}^{d'}$. If f is chosen to be a MLP, one obtains a so-called *DeepSets* [12] model. Next, we look at GNNs that do utilize graph structure.

Most graph convolution approaches are based on the principle of aggregating vertices that are spatially related, typically by being direct neighbors. The so-called graph convolutional network (GCN) convolution [23], for example, updates the feature vector of each vertex by computing the mean of the features of its neighbors. Xu et al. [13] show that this approach has a limited *discriminative power*, i.e., it cannot distinguish some classes of non-isomorphic graphs. To address this limitation, they propose the more powerful graph isomorphism network (GIN) architecture.

Recently, multiple approaches going beyond the discriminative power of GIN have been proposed [14, 24, 25]. The so-called 2-WL-GNN architecture [14], for example, is based on the 2-dimensional (Folklore) Weisfeiler-Lehman (WL) graph isomorphism test [26]; it is provably more powerful than GIN.

2) Unsafe Usage Classification: The architecture of our `unsafe` usage classification model follows the standard GNN structure for graph-level prediction tasks, i.e., it consists of a sequence of graph convolution layers, followed by a graph pooling layer, which produces a vector embedding for a given CFG. This embedding is a summary of the encoded information and reduces the size of the feature map. This pooled embedding vector is then concatenated, to compensate for the loss introduced by the pooling layer, with two additional vectors that encode the following information:

Usage Location: A Go function may contain multiple `unsafe` usages with different `WHAT` and `WHY` labels. However, in the pooled graph embedding vector, all statements that contain `unsafe` are merged and can no longer be distinguished. We address this issue by concatenating the pooled vector with the convolved feature vector of the vertex representing the Go statement, which contains the `unsafe` usage that should be labeled.

Context Type: Additionally, we append a one-hot context type vector, which encodes whether the `unsafe` usage occurs in a function declaration, a global variable, or a type definition. Since global variable and type definitions are both represented as single `declaration` statement vertices, they generally cannot be distinguished from each other and from functions without parameters and return types whose body only contains a single variable declaration. The context type vector allows the model to distinguish these cases. Finally, the concatenated embedding vector is fed into a MLP, followed by two parallel fully-connected layers with softmax activations, which produce two label probability distributions for the `WHAT` and `WHY` labels. We present this architecture in Fig. 1 and mark the `unsafe` usage with red. The concatenated embedding vector is represented by the edge after the concatenation icon in the right half of the Figure.

For the evaluation in Section IV, we created four variants of this architecture using the following types of graph convolutions: DeepSets [12], GIN [13] and 2-WL-GNN [14]. To determine the importance of the context in which an `unsafe` usage occurs, we additionally built a simple baseline MLP model, which only gets the feature vector of the vertex representing the statement to be classified.

We train our models with the Adam optimizer [27] and the sum of the cross-entropies for both labels as the loss function; i.e., the optimization target is to maximize the predicted probabilities of the correct labels for each usage. Modern neural network classifiers often tend to be overly confident, i.e., the model's accuracy is lower than the average probability it assigns to its top-1 predictions [28]. To address this issue, we calibrate the predicted probabilities for `WHAT` and `WHY` independently via temperature scaling [29, 28].

We do not only output the `WHAT` and `WHY` labels with the highest predicted probability but instead output a set of labels for each label category. The rationale is to account for the fact that correct labels for a given `unsafe` usage are not always obvious, even to a human expert. The predicted sets can serve as a preselection that assists the user in determining the most plausible `WHAT` and `WHY` labels. Each prediction set is created by selecting the top-k classes until the probability mass of the set exceeds a certain threshold. The threshold is chosen via *inductive conformal prediction* using the so-called *generalized inverse quantile nonconformity score* [18]. This approach provably guarantees that the predicted `WHAT` and `WHY` sets each contains the respective true label for a given usage with a probability of at least $1 - \varepsilon$, where $\varepsilon \in [0, 1]$ is a significance level that can be freely chosen; we use the common default $\varepsilon = 0.1$. In addition to this validity guarantee, the sizes of the conformal prediction sets are adaptive; if the model is uncertain about the label for a given usage, it will produce a large prediction set. Likewise, a small set is produced if the model is certain about the true label.

IV. Implementation and Evaluation

We implemented our approach as a self-contained tool, UnGoML, which provides functionality to both identify and quantify `unsafe` snippets (go-geiger [2]) along with the classification for `unsafe` usages (Fig. 1). UnGoML expects a Go project, e.g., local or a GitHub URL, as input and returns a report of `unsafe` usages within the Go project, including the prediction of `WHAT` and `WHY`. To generate this report, UnGoML runs go-geiger upon the project. Afterward, we pass all identified `unsafe` usages to the classifier, which predicts for each snippet `WHAT` and `WHY` labels. These predictions are combined with the results of go-geiger to generate a report of all `unsafe` usages within the given project.

In Subsection IV-A we describe the studied dataset, followed by our experimental setup in Subsection IV-B, and answer the following three research questions in Subsection IV-C

RQ1: **What is the impact of the context in predicting `WHAT` and `WHY` of `unsafe` usages?** The GNN models get the context of a usage as input, while the MLP baseline only considers a single vertex.

RQ2: **What is the impact of control- and data-flow for classifying `unsafe` usages?** Control- and data-

flows are commonly used, e.g., in static analyses, to reason about code, and are effective for deep learning vulnerability detection [10, 15, 17].

RQ3: **How relevant are different vertex features for the classification of `unsafe` usages?** The vertex labels in our enriched CFG representation encode various aspects of a code snippet. We want to determine the importance of the different vertex label categories, e.g., DATATYPES or FUNCTIONS.

We also present the predictions for Listing 1 in Subsection IV-D, and discuss in Subsection IV-E how UNGOML can be used in practice.

A. Studied Dataset

We chose our dataset [2] over the one of Costa et al. [3] for the following reasons. First, with 1,400 entries, it is much larger than the 270 labeled entries of Costa et al. [3]. Second, the labels are provided in two dimensions instead of one, thus providing a more fine-granular representation of the `unsafe` usage. Further, our comparison of the usage patterns (cf. Table I) confirms that we identified more diverse `unsafe` usage patterns. Third, we [2] labeled `unsafe` usages on a statement-level instead of a file-level. Thus, we can predict usages more fine-granular. Fourth, the analysis of Costa et al. [3] focuses only on `unsafe` usages within the application code without considering dependencies. Thus, it ignores a common source of many `unsafe` usages [2] and vulnerabilities [30, 31]. To train our model, we used the dataset as-is. The labels are derived from the 10 projects of the top-500 starred GitHub projects with the most `unsafe` usages. For more details about this data set, we refer to our paper [2].

B. Experimental Setup

To train and evaluate different model architectures, we use an existing manual labeled data set of `unsafe` usages [2] (more details Section II-B). We randomly split the dataset into ten stratified folds of equal size [32], preserving the joint distribution of the `WHAT` and `WHY` label dimensions in each fold. Each model is independently tuned, trained, and evaluated ten times using each bin once as test data and in the other iterations as training/validation data. All following results are averages over the ten iterations.

In each iteration, we further subdivide the training/validation data into a 90% training split and a 10% validation holdout split. As mentioned in Section III-B2, we use the Adam optimizer [27] to minimize the sum of the cross-entropy losses for both labels. The learning rate is fixed at 0.001. To tune the hyperparameters of each model, we use Hyperband [33] with a reduction factor of 3 and a maximum epoch count of 200. As the hyperparameter optimization objective, we use the joint top-1 accuracy on the validation split, i.e., the proportion of validation instances for which the `WHAT` and `WHY` labels with the highest predicted probability are *both* correct. The explored hyperparameter space is shown in Table II. The activations and widths of the convolution and MLP layers are tuned independently. The `softmax` pooling treats one vertex feature dimension as a weight distribution logic to compute the weighted mean of the remaining dimensions. Batch normalization is only applied after convolutions, dropout only between MLP layers.

TABLE II: Explored hyperparameter space.

Parameter	Values
Depths	Conv. $\in \{2, \ldots, 6\}$, MLP $\in \{1, 2, 3\}$
Activations	$\{$`relu`, `sigmoid`, `tanh`, `elu`$\}$
Layer Widths	$\{n \in \mathbb{N} \mid n \pmod{32} \equiv 0 \wedge n \leq 512\}$
Pooling	$\{$`sum`, `mean`, `max`, `min`, `softmax`$\}$
Regularization	Batch Norm $\in \{$yes, no$\}$, Dropout $\in \{0, \frac{1}{2}\}$

TABLE III: Unsafe usage combined mean test classification accuracies (in %) and the set sizes for conformal set prediction.

	Top-1 Acc.	Top-3 Acc.	Conformal Set `WHAT` Size	Conformal Set `WHY` Size	Conformal Set Acc.
Majority	28.9 ± 0.0	48.9	8.0 ± 0.0	5.8 ± 0.0	88.4
MLP	73.8 ± 0.4	87.3	2.5 ± 0.1	2.6 ± 0.1	92.5
DeepSets	79.8 ± 0.4	91.5	2.1 ± 0.1	1.9 ± 0.1	93.2
GIN	78.0 ± 0.5	91.8	2.2 ± 0.1	2.0 ± 0.1	93.1
2-WL-GNN	79.3 ± 0.5	91.6	2.2 ± 0.1	1.9 ± 0.1	93.2

After hyperparameter selection, the model is trained three times to estimate the performance variance caused by random weight initialization. For each training repeat, we use a maximum epoch limit of 1,000 in combination with early stopping if the validation loss does not decrease for 100 epochs. We use the validation split as the calibration data for temperature scaling [29, 28] and conformal prediction. To evaluate each trained and calibrated model, we use the top-1, top-3, and conformal set accuracy, plus the mean conformal set size on the test data. We aggregate the results for the three repeats by computing the mean and standard deviation for each metric. Those results for each of the ten outer iterations are further aggregated by computing the expected value and standard deviation of their mean.

C. Model Comparison and Ablation Study

Table III presents the aggregated results we obtained for the evaluated models trained with all feature dimensions (Section III-A3). We compare the accuracies obtained for all models relative to a majority classifier that predicts the distribution of `WHAT` and `WHY` in its training data. Overall, we find that all four types of models significantly outperform the baseline majority classifier by at least 45%, indicating that they successfully learn generalizable correlations between CFG properties and the target labels. Also, all context-aware models perform similarly and have consistently higher accuracies than the MLP models in top-1 (6%) and top-3 (4%).

Answer to Research Question 1

The three context-aware models significantly outperform the single vertex MLP model, showing that the context is important for `unsafe` classification.

The DeepSets model achieves essentially the same mean results as 2-WL-GNN without considering control-flow structure

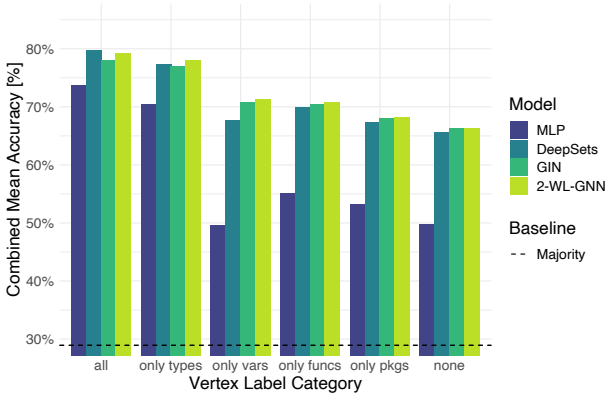

Fig. 3: Combined Top-1 Prediction Accuracy for different vertex feature sets. ALL shows the results for CFGs with the full set of vertex features. For the remaining, all (NONE) or all but one infinite vertex label category are removed.

and variable usage dependencies. Note, that only 2-WL-GNN considers edge labels, while GIN treats all edges equally. This architecture generally performs slightly worse than the other two context-aware models.

Answer to Research Question 2

For `unsafe` classification, the impact of control- and data-flow is negligible, as DeepSets achieves essentially the same accuracies as 2-WL-GNN and GIN.

Figure 3 shows the aggregated results we obtained for the evaluated models when trained with different subsets of the feature dimensions (Section III-A3) and models. The block ALL contains the results for the models trained with the complete set of features. In the NONE block, all variable vertices and all vertex labels, except those from the categories STATEMENT TYPES and SELF-REFERENCES are omitted. In the other blocks, the NONE block is extended by exactly one infinite vertex label category. The block ONLY VARS includes the variable vertices, ONLY TYPES adds the DATATYPES labels, and ONLY FUNCS enters the FUNCTIONS and OPERATOR labels. Lastly, ONLY PKGS includes the PACKAGES labels. As in Table III, we added the majority classifier as a baseline.

Comparing different feature subsets, we find that the DATATYPES labels are the most important – the block ONLY TYPES comes closest to the models trained with the complete feature set. This is plausible since the DATATYPES label category contains, among others, the `unsafe.Pointer` and `uintptr` labels. The other feature subsets are generally much less informative for classifying `unsafe` usages.

Answer to Research Question 3

The most important features for the `unsafe` classification are the DATATYPES.

D. Example Prediction: Listing 1

We elaborate on the predicted labels for the example presented in Listing 1 to gain insights into concrete predictions. We consider the labels (`WHAT`: *pointer-arithmetic*, `WHY`: *serialization*) defined by Lauinger et al. [2] as the expected ones and use the term alternative labels to refer to the two additional labels (`WHAT`: *cast*, `WHY`: *efficiency*) discussed in Section II-C.

The baseline MLP model, which only considers the single statement vertex containing the `unsafe` usage that is to be classified, successfully predicts the expected labels for `WHAT` and `WHY` as its top-1 predictions. Within the conformal prediction, the alternative label for `WHAT` has the second-highest probability. However, the alternative label for `WHY` is not included in the conformal set. We attribute this to the fact that the information that hints at *efficiency* is not present within the single vertex feature vector given to the MLP classifier.

The top-1 prediction of the DeepSets and GIN models include an expected label and an alternative label for `WHAT` and `WHY`. Both of these labels for `WHAT` and `WHY` are included in the top-2 predictions for the models. For DeepSets, we observe a larger set size than those predicted by MLP. We attribute this observation to the additional information that slightly misled the DeepSets model. For example, for `WHAT` the data type `bool` used in Line 3 is a positive indicator[5] for two of the predicted labels, namely *cast-basic* and *cast-bytes*. As the model is unaware of the control flow, it cannot distinguish if the type influences the `unsafe` usage.

The 2-WL-GNN model returns both labels for `WHAT` and `WHY` as its top-1 prediction. Additionally, this model was the only one predicting no further labels in its conformal sets.

E. Applications of UNGOML

We discuss two exemplary applications of UNGOML: (a) helping security auditors to identify fragments with security-relevant `unsafe` usages and (b) assisting developers in refactoring code to replace usages of `unsafe` that mimic generics with the generics language construct recently added to Go.

We use the confusion matrices in Fig. 4 to estimate the effectiveness of UNGOML when applied to the above use cases. These matrices present the recall of each label (in the diagonal) and the percentage of falsely predicted labels (other values in the row of the label). We present the confusion matrices for the MLP and 2-WL-GNN models; the 2-WL-GNN model is considered as a representative of all (similarly performing) context-aware models and the MLP is the non-context-aware model.

1) Security Audit: When conducting a security audit, UNGOML can help auditors to prioritize code fragments that require a manual review by focusing their attention on the most security-relevant `unsafe` usages, i.e., usages labeled as *casts*, *pointer-arithmetic*, and *memory-access* in `WHAT`.

The top-1 recall for the MLP model to classify *memory-access* and *pointer-arithmetic* is 15% and 47%; the 2-WL-GNN model achieves a much better recall of 37% and

[5]It is among the top-3 important features returned by Grad ⊙ Input [34].

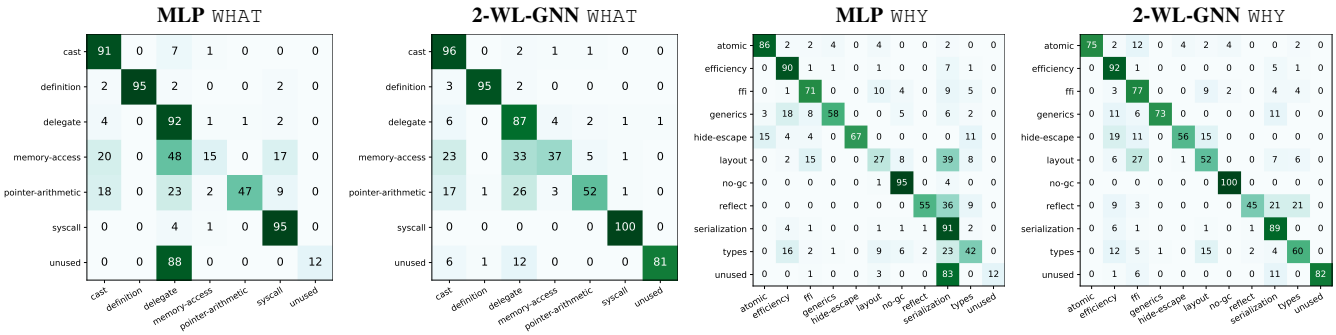

Fig. 4: Row-normalized sum over the confusion matrices on the test splits using the complete set of vertex features. The horizontal axis corresponds to the predicted labels, the vertical axis to the true labels. Values are given as percentages.

52% – hence, we focus on this model in the following. As already stated, 2-WL-GNN predicts *memory-access* with a 37% accuracy; 5% of the falsely predicted labels for *memory-access* are classified as *pointer-arithmetic*. The most confusing label is *delegate* with 33%. For *pointer-arithmetic* we have a 3% confusion with *memory-access* and 26% confusion with *delegate*. Thus, 2-WL-GNN narrows down an `unsafe` finding to *memory-access*, *pointer-arithmetic*, or *delegate* with a combined top-1 recall of 75% to 81%. As all other labels have quite high detection rates (81% to 100%), one can easily rule out other `unsafe` usages. The only other relevant source of confusion are *cast*s, which account for 23% to 17%; this is, however, not a problem, as casts are also relevant for security audits. For actual cast operations, 2-WL-GNN achieves a recall 96%.[6]

2) Refactoring Generics: Developers can use UNGOML to identify `unsafe` usages that mimic generics (labeled *generics* in `WHY`). For each usage labeled as such, they can decide if and how to replace it with the newly introduced language feature for generics. The recall for label *generics* varies between 59% for MLP and 73% for 2-WL-GNN model. While a 73% recall is already quite high, the effective recall for real-world generics detection is even higher. In practice, the absence of language-integrated generics is often compensated with code duplication. Thus, it is sufficient to detect one of the duplicates to refactor all `unsafe` usages that mimic generics.

Summary

We conclude that our approach is practical for the discussed applications. Further, 2-WL-GNN, UNGOML's default model, performs well in both use cases.

3) Generalizability: In addition to the two above-mentioned concrete use cases, UNGOML can be used in other contexts. The used GNN models can cover other types of `unsafe` usages because our implementation can be used to train them with other Go datasets that contain other types of `unsafe` usages or other kinds of vulnerabilities. Further, one can replace or extend parts of our pipeline, such as the implementation to create our code representation, to integrate data sets in other languages.

[6]It can even distinguish types of casts, e.g., structs, bytes, basics, etc.

V. THREATS TO VALIDITY

A potential threat to **internal validity** is the correctness of our implementation, evaluation, and CFG representation. We only included a small subset of the information available for a given `unsafe` usage and exclude call sites and callees in the CFG. Thus, not all relationships between different elements are fully represented. We decided against exploring more fine-grained representations as our experiments to encode the abstract syntax tree of statements did not improve the classification accuracy. Further, more information increases the graph sizes and the time required to train a model. Further, currently the representation is intraprocedural. Also, we ignore natural language information from comments, even though a human annotator might consider them when labeling a usage.

The selection of our dataset is a potential threat to **construct validity**. We used the dataset from our previous work [2] to train and evaluate our classifier. The dataset by Costa et al. [3] is not selected as it provides labels only on a file level which is not suitable for our research. Further, for a fair comparison we did not compare UNGOML against the existing static analyzers that can only detect `unsafe` usages without any classification or only a few usage patterns.

The generalizability of our findings is affected by our threats to **external validity**. For training and verification, we rely on the correctness and representativeness of the labels we created previously [2]. As these labels subsume the ones defined by Costa et al. [3] and both were derived from usages obtained from popular GitHub projects, we believe that we chose a suitable data set as ground truth.

Further, our classifier is trained on the `unsafe` package of Go 1.16 and below and does not reflect recent changes in the package. Concretely, Go 1.17 (released August 2021) introduced two new functions [19] (Section II-A). These changes were not labeled, as the labels were created before the release. However, we believe this will not affect our accuracy as the vast majority of usages are for the type `unsafe.Pointer` [2, 3] rather than the `unsafe` functions.

Lastly, our comparison of different graph convolutions is not comprehensive and only considers existing approaches. Namely, we chose the DeepSets, GIN, and 2-WL-GNN architectures as representatives of structure unaware, 1-WL-bounded, and higher-order convolutions, respectively. Due to the limited influence of CFG edges on the overall model performance, no other types of convolutions were included in the evaluation. However, we cannot exclude the possibility that a specialized graph convolution operator might improve the prediction quality further.

VI. Related Work

Unsafe API Usages. Previous studies on `unsafe` usages mostly focused on detecting the usages and classifying the usages manually [2, 3]. Unlike our work, they relied on a time-consuming, challenging, and error-prone manual validation (see Table I). The moderate agreement of the Cohen-kappa score [35] of 0.65 reported by Costa et al. [3] confirms the difficulty in labeling the different usages precisely.

Previous studies on `unsafe` usages in Java focused on detecting usages and their patterns [4] and did not provide a classification tool. For Rust, previous studies concentrated on understanding `unsafe` usages empirically [36, 37, 5]. Furthermore, a survey revealed that most Rust developers use `unsafe`, e.g., to use foreign-function-interfaces or interact with hardware [38]. The RustBelt project provides Rust programmers with formal tools for verifying safe encapsulation of `unsafe` [6]. Static analyses were developed for a subset of definite `unsafe` patterns causing bugs [2, 37, 39]. However, the analyses only identify a subset of problems and miss to provide support to classify all `unsafe` usage patterns.

Static Analyses to Detect `unsafe` Usages. Existing analyses for `unsafe` usages in Go, are either simple linters to detect `unsafe` usages [2, 40], cover only a few selected patterns [2, 7], or can detect violations only during runtime [8]. Mature static analyses frameworks such as Doop [41], Soot [42], or Opal [43] do not exist for Go. Thus, current analyses are limited to taint analyses [44, 45] or simple AST-based analyses [7, 40]. While a taint analysis is not suitable for the task at hand, AST-based analyses are very simple. We acknowledge that one could leverage the AST-based analyses to implement a few patterns for categories such as *definition* and *delegate*. Nonetheless, we doubt that this would work for patterns such as *efficiency* or *reflection* completely. However, we would be interested in future work exploring this line of research..

Deep Learning for Vulnerabilities. As far as we know, we are the first who classify `unsafe` usages. Thus, we discuss deep learning solutions for vulnerability detection as a related problem. Recent work on the classification of vulnerabilities, e.g., buffer overflows, has shown the effectiveness of GNN architectures [17, 16, 15, 9, 10]. The majority of deep learning classifiers are trained on binary decisions, namely vulnerable or non-vulnerable code [10, 46]. Recently, Wang et al. [9] predicted the vulnerability type and showed that the precision drops compared to a binary classifier. Our classifier achieves a top-1 accuracy similar to their top-3 accuracy.

Although many approaches for vulnerability prediction support developers and auditors, many provide a coarse-grained prediction level such as file, function, or method-level [17, 10, 15]. Thus, solutions to reduce the number of lines to be inspected are proposed, e.g., by inspecting the subgraph that influences the prediction the most [47]. Recently, more fine-granular predictions on line-level are suggested [47]. However, Li et al. [47] tokenize code rather than building graphs and leveraging different GNNs architectures.

As previous works [9, 46] that use GNNs, we build our intermediate representation by parsing a graph from source code enriched with relevant information for our classification. As we aim to solve a different problem, we have to include additional information, such as data types, instead of *LastUse* information [9]. In contrast to Duan et al. [46], our feature vector with 594 possible vertex labels is more expressive than their proposed 144 labels.

VII. Conclusion

In this paper, we presented the first classifier, UnGoML, for `unsafe` usages in Go. UnGoML helps to understand the actual usage (`WHAT`) and the underlying purpose (`WHY`) of using this escape hatch from memory safety. We encoded the `unsafe` code snippets as enriched CFGs and classified them with GNNs. To further understand the relevance of features, we varied the included features, e.g., by only including the variables or only the datatype information. With the full set of features, we achieve a mean top-1 accuracy of about 88% and 87% for `WHAT` and `WHY`, respectively, with the 2-WL-GNN architecture. Furthermore, we show that a set-value conformal prediction classifier returns on average 2 labels with a mean accuracy of 93%. Thus, our classifier is suitable to effectively support developers and auditors to identify and refactor `unsafe` usages, e.g., to replace `unsafe` with generics or avoid potentially vulnerable `unsafe` usages.

In future work, UnGoML can be leveraged for automatic large-scale refactoring and auditing tasks. Further, our methodology and insights gained can be transferred to other domains, such as API misuses in general, by adapting the data set and our implementation. In addition, our classifier can be used for a comparison of static analyses that can detect the discussed `unsafe` usage patterns.

Acknowledgment

We are grateful for the valuable feedback that we received from the reviewers that helped us to improve the paper. We want also to thank Antonio Zhu for his work on wrapping two of our tools into UnGoML.

This paper is based on work funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297 and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] NSA Media Relations, "Nsa releases guidance on how to protect against software memory safety issues," 2022, latex update: Nov, 10 2022, accessed: Mar, 17 2023. [Online]. Available: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/

[2] J. Lauinger, L. Baumgärtner, A.-K. Wickert, and M. Mezini, "Uncovering the hidden dangers: Finding unsafe go code in the wild," in *19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020.

[3] D. E. Costa, S. Mujahid, R. Abdalkareem, and E. Shihab, "Breaking type-safety in go: An empirical study on the usage of the unsafe package," *Transactions on Software Engineering (TSE)*, 2021.

[4] 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 *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2015.

[5] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2020.

[6] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," in *Proceedings of the ACM on Programming Languages (POPL)*. ACM, 2018.

[7] The Go Authors, "vet command - cmd/vet - go packages," 2023, latest update Mar, 7 2023, accessed: Mar, 17 2023. [Online]. Available: https://pkg.go.dev/cmd/vet

[8] M. Dempsky, "cmd/compile: add -d=checkptr to validate unsafe.pointer rules," 2019, latest update Oct 17, 2019, accessed: Mar, 17 2023. [Online]. Available: https://go-review.googlesource.com/c/go/+/162237

[9] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection," *Transactions on Information Forensics and Security*, 2021.

[10] T. Sonnekalb, T. S. Heinze, and P. Mäder, "Deep security analysis of program code: A systematic literature review," *Empirical Software Engineering*, vol. 27, Jan. 2022.

[11] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *19th International Conference on Mining Software Repositories (MSR)*. ACM, 2022.

[12] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Póczos, R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[13] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *7th International Conference on Learning Representations (ICLR)*, 2019.

[14] C. Damke, V. Melnikov, and E. Hüllermeier, "A Novel Higher-order Weisfeiler-Lehman Graph Convolution," in *12th Asian Conference on Machine Learning (ACML)*, ser. Proceedings of Machine Learning Research (PMLR), vol. 129, 2020.

[15] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *Transactions on Software Engineering (TSE)*, 2021.

[16] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, 2021.

[17] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[18] Y. Romano, M. Sesia, and E. J. Candès, "Classification with valid and adaptive coverage," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[19] Golang, "Unsafe," aug 2022, latest update: Mar, 7 2023, accessed: Mar, 17 2023. [Online]. Available: https://pkg.go.dev/unsafe

[20] gogoprotobuf, "Gogo/protobuf - pointer_unsafe.go," 2018, latest commit 9c3ad97, accessed: Mar, 17 2023. [Online]. Available: https://github.com/gogo/protobuf/blob/b03c65ea87cdc3521ede29f62fe3ce239267c1bc/proto/pointer_unsafe.go

[21] ——, "Gogo/protobuf - table_marshal.go," 2019, latest commit 09ab773, accessed: Mar, 17 2023. [Online]. Available: https://github.com/gogo/protobuf/blob/b03c65ea87cdc3521ede29f62fe3ce239267c1bc/proto/table_marshal.go

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[23] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations (ICLR)*, 2017.

[24] G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein, "Improving graph neural network expressivity via subgraph isomorphism counting," *Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[25] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman, "Provably powerful graph networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*

(NeurIPS), 2019.

[26] J. yi Cai, M. Fürer, and N. Immerman, "An optimal lower bound on the number of variables for graph identification," *Combinatorica*, vol. 12, no. 4, 1992.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations (ICLR)*, 2015.

[28] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *34th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research (PMLR), vol. 70, 2017.

[29] J. C. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," *Advances in large margin classifiers*, 1999.

[30] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies," *Transactions on Software Engineering (TSE)*, vol. 48, no. 5, May 2022.

[31] M. Zimmermann, C.-A. Staicu, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," Aug. 2019.

[32] T. Hall and D. Bowes, "The state of machine learning methodology in software fault prediction," in *11th international conference on machine learning and applications*, vol. 2. IEEE, 2012.

[33] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research (JMLR)*, vol. 18, 2018.

[34] B. Sanchez-Lengeling, J. N. Wei, B. K. Lee, E. Reif, P. Wang, W. W. Qian, K. McCloskey, L. J. Colwell, and A. B. Wiltschko, "Evaluating attribution for graph neural networks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[35] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, 2012.

[36] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[37] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *41st Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 763–779.

[38] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: rust as a case study," in *17th Symposium on Usable Privacy and Security (SOUPS)*, 2021.

[39] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *28th Symposium on Operating Systems Principles (SOPS)*. ACM, 2021.

[40] G. Murphy, "G103: Use of unsafe block - secure go," 2020, accessed: Mar, 17 2023. [Online]. Available: https://securego.io/docs/rules/g103.html

[41] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2009.

[42] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. USA: IBM Corp., 2010.

[43] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. ACM, 2019.

[44] E. Bodden, K. I. Pun, M. Steffen, V. Stolz, and A.-K. Wickert, "Information flow analysis for go," in *International Symposium on Leveraging Applications of Formal Methods (ISOLA)*. Springer, 2016.

[45] W. Li, S. Jia, L. Liu, F. Zheng, Y. Ma, and J. Lin, "Cryptogo: Automatic detection of go cryptographic api misuses," in *38th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2022.

[46] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities," in *28th International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[47] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2021.