BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EVALUATING RELIABILITY OF STATIC ANALYSIS RESULTS USING MACHINE LEARNING

URČENÍ SPOLEHLIVOSTI VÝSLEDKŮ STATICKÉ ANALÝZY POMOCÍ STROJOVÉHO UČENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TOMÁŠ BERÁNEK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

157228

Institut:	Department of Intelligent Systems (DITS)
Student:	**Beránek Tomáš, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Určení spolehlivosti výsledků statické analýzy pomocí strojového učení**
Category:	Artificial Intelligence
Academic year:	2023/24

Assignment:

1. Get acquainted with Infer, a tool for static analysis and bug finding in software.
2. Investigate options of applying machine learning algorithms in the context of code analysis.
3. Obtain a data-set containing issues reported by Infer accompanied with the information whether they represent a true positive or not.
4. Design and implement a system that converts the dataset obtained in Step 3 into a format that can be processed by graph neural networks.
5. Propose and implement an approach based on graph neural networks (using the dataset obtained in Step 4) whose goal will be to assess the likelihood that an issue reported by Infer represents a true positive.
6. Evaluate your solution on at least 2 different open-source projects.
7. Summarize and discuss the achieved results and their possible further improvements.

Literature:

- Facebook Infer: https://fbinfer.com/
- Cao, Sicong, et al. "Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection." *Information and Software Technology* 136 (2021): 106576.
- Y. Zheng et al., "D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis," 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 111-120.

Requirements for the semestral defence:
The first three points of the assignment and at least the beginning of work on Point 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Consultants:	Grác Marek, Mgr., Ph.D. Malík Viktor, Ing.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

The Meta Infer static analyzer is a tool for detecting various types of errors in source code. However, its results contain more than 95 % of false alarms. This thesis proposes a solution that ranks Infer's reports using Graph Neural Networks (GNNs) based on the likelihood of being a real error, thus mitigating the issue with false alarms. The system consists of a training pipeline, which converts the D2A dataset – a set of labeled reports from Meta Infer – into Extended Code Property Graphs (ECPGs) and GNN models trained on these ECPGs. Experimental results indicate that the developed GNN models can match, and in some cases even surpass, existing models developed by strong industrial teams. Moreover, these existing solutions are closed source, making the solution developed in this thesis a promising open-source alternative.

Abstrakt

Statický analyzátor Meta Infer je nástrojem pro hledání různých typů chyb ve zdrojovém kódu. Jeho výsledky však obsahují více než 95 % falešných hlášení. Tato teze navrhuje řešení, které řadí hlášení od Meta Inferu pomocí grafových neuronových sítí (GNN) podle pravděpodobnosti, že se jedná o skutečnou chybu, a redukuje tak problém s falešnými hlášeními. Systém se skládá z trénovací části, která převádí datovou sadu D2A – sadu roztříděných hlášení z Meta Inferu – na rozšířené grafy vlastností kódu (ECPG) a z modelů GNN natrénovaných na ECPG grafech. Výsledky experimentů ukazují, že vytvořené modely GNN mohou konkurovat a v některých případech dokonce překonat existující řešení vyvíjené silnými průmyslovými týmy. Tato existující řešení mají navíc uzavřený zdrojový kód, a tak řešení vytvořené v této tezi poskytuje slibnou alternativu s otevřeným zdrojovým kódem.

Keywords

Static analysis, Meta Infer, deep learning, graph neural networks, false alarm detection, vulnerability detection, code property graphs, LLVM internal representation, Joern, LLVM Slicer, program slicing, graph representation construction, source code analysis, D2A dataset, graph D2A dataset, extended code property graphs.

Klíčová slova

Statická analýza, Meta Infer, hluboké učení, grafové neuronové sítě, detekce falešných hlášení, detekce zranitelností, grafy vlastností kódu, interní reprezentace LLVM, Joern, LLVM Slicer, prořezávání programů, konstrukce grafové reprezentace, analýza zdrojového kódu, dataset D2A, grafový D2A, rozšířené grafy vlastností kódu.

Reference

BERÁNEK, Tomáš. *Evaluating Reliability of Static Analysis Results Using Machine Learning.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Statická analýza je často využívanou technikou pro hledání chyb v průběhu vývoje softwaru. Některé statické analyzátory lze použít i na nedokončený kód, což umožňuje odhalení chyb již ve velmi raných fázích vývoje, dokonce ještě před spuštěním testů. Tyto nástroje však často trpí vysokým počtem falešných hlášení. Pokud je množství těchto falešných hlášení příliš vysoké, stávají se tyto nástroje v praxi téměř nepoužitelnými, protože kontrola hlášení je příliš nákladná. Proto je věnováno mnoho úsilí automatizované detekci falešných hlášení.

Tato diplomová práce se zaměřuje na statický analyzátor Meta Infer. Jedná se o vysoce škálující, mezi procedurální, open-source nástroj pro analýzu zdrojových souborů v jazycích C/C++/C#/Obj-C a Java. Infer dokáže detekovat chyby jako dereference nulových ukazatelů, mrtvé úložiště, neinicializované hodnoty, přetečení proměnných a mnoho dalších typů chyb. Oproti jiným statickým analyzátorům se vyznačuje snadným používáním – jeho vstupem jsou kompilační příkazy, které kompilují analyzované zdrojové soubory. Přesto má tento nástroj své nevýhody, přičemž hlavní z nich je vysoký počet falešných hlášení. V experimentech provedených v autorově bakalářské práci bylo zjištěno, že až 90 % hlášení je falešných. Toto číslo se zvyšuje na více než 95 %, pokud nejsou zohledněny chyby typu mrtvého úložiště, které jsou samy o sobě poměrně běžné a neškodné.

Hlavním přínosem této diplomové práce je návrh a implementace systému pro hodnocení hlášení nástroje Meta Infer. Vyvinutý systém dokáže řadit hlášení podle pravděpodobnosti, že se jedná o skutečnou chybu (tj. pravdivé hlášení), čímž řeší problém s velkým množstvím falešných hlášení a činí Infer prakticky použitelnějším, protože současné procento falešných hlášení je příliš vysoké.

Systém pro řazení hlášení je založen na grafových neuronových sítích (GNN), které v posledních letech získaly na popularitě pro úkoly související se zdrojovým kódem, protože mnoho vlastností kódu lze přirozeně vyjádřit pomocí grafů – grafy toků řízení, abstraktní syntaktické stromy, grafy závislostí dat a mnoho dalších. Pro trénování modelů GNN je nezbytná datová sada. Tato diplomová práce využívá datové sady D2A, která obsahuje roztříděná (skutečná vs. falešná) hlášení od Inferu ze šesti open-source projektů. D2A obsahuje vzorky ve formě textu, které je třeba převést do formy grafů. Z tohoto důvodu byl vytvořen tréninkový proces, který generuje Graph D2A – D2A převedené do formy grafů. Tréninkový proces doplňuje existující techniky vytváření grafů o informace o podmíněném překladu, který se v praxi často vyskytuje.

Vzorky v Graph D2A nelze přímo použít pro trénování modelů GNN; nejdříve na nich musí být proveden výběr a transformace příznaků. Po výběru a transformaci příznaků jsou grafy optimalizované a převedené do formátu navrženého v této diplomové práci – rozšířené grafy vlastností kódu (ECPG), které obohacují stávající grafy vlastností kódu (CPG) o grafy volání, datové typy a řadu dalších informací. CPG jsou běžně používaným formátem grafů pro detekci zranitelností ve zdrojovém kódu pomocí GNN.

Vytvořené modely byly trénovány na trénovací sadě tří projektů z D2A, jmenovitě httpd, libtiff a nginx. Vyhodnocení modelů probíhalo na testovací sadě stejných projektů. Experimentální výsledky ukazují, že vytvořené modely GNN mohou konkurovat a v některých případech dokonce překonat nejlepší stávající řešení, která jsou vyvíjena silnými průmyslovými týmy. Tyto výsledky dokazují, že vytvořené modely jsou slibnou open-source alternativou k porovnávaným existujícím řešením, která všechna mají uzavřený kód.

Modely byly také testovány pomocí křížové analýzy – model je testován na jiném projektu, než na kterém byl trénován. Modely se pro tuto výzvu ukázaly jako nedostatečné, což pouze vyzdvihuje obtížnost křížové analýzy v této oblasti výzkumu, jelikož žádné z porovnávaných existujících řešení taktéž pro křížovou analýzu nefunguje.

Posledním přínosem této diplomové práce je inferenční proces, který umožňuje spustit Infer analýzu, generovat ECPG pro každé hlášení a nakonec řadit hlášení pomocí vytvořených modelů GNN pro libovolný software v jazyce C (a podmnožině C++). Princip inferenčního procesu staví na autorově bakalářské práci, která se zabývala automatizací Infer analýzy. Nicméně inferenční proces zůstává nevyužitý, kvůli zatím nefunkční křížové analýze.

Předběžné výsledky této diplomové práce byly publikovány na konferenci Excel@FIT'24, kde obdržely ocenění od odborného panelu.

Evaluating Reliability of Static Analysis Results Using Machine Learning

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by Mgr. Marek Grác, Ph.D. and Ing. Viktor Malík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tomáš Beránek
17. května 2024

Acknowledgements

I would particularly like to thank my supervisor Tomáš Vojnar for numerous helpful pieces of advice, not only for this thesis but also for my studies. I also wish to express my thanks to Marek Grác for valuable advice in the area of artificial intelligence and Viktor Malík especially for arranging the possibility of working on this topic and also for helpful advice on program slicing. Further, I thank my colleagues Tomáš Dacík, Dominik Harmim, Daniel Marek, and Lucie Svobodová for helpful discussions about Infer.

Finally, I acknowledge the financial support received from Red Hat and projects H2020 ECSEL Valu3s, GACR AIDE 23-06506S, and IGA FIT-S-23-8151.

Contents

Chapter 1

Introduction

Static analysis is a widely used technique for finding errors during *software development*. *Static analyzers* can also be deployed on code that is not yet finished, making it possible to detect errors in the early stages of the development, even before tests can be run. However, static analyzers often suffer from a high number of *false positives* (i.e., *false alarms*). If the percentage of false positives is too high, these tools are almost unusable in practice. Therefore, a lot of effort is devoted to the automatic detection of false positives.

This thesis focuses on the Meta Infer static analyzer. It is a highly *scalable*, *interprocedural*, *open-source* tool for analyzing C/C++/C#/Obj-C, and Java source files. Infer can detect *null pointer dereferences*, *dead stores*, *uninitialized values*, *deadlocks*, *data races*, *variable overflows*, and many other types of errors. Compared to other static analyzers, it is characterized by its ease of use – its input consists of the compilation commands that compile the analyzed source files. Although Infer has been successfully used in practice by a number of companies (including Meta), it does have its disadvantages, and the main one is the high number of false positives. From experiments conducted in the author's bachelor's thesis, it was found that up to 90 % of the reports are false positives. This number increases to more than 95 % if errors of the dead store type, which are relatively common and harmless by themselves, are not considered.

The main contribution of the thesis is the design and implementation of a *report ranking* system for the Meta Infer tool. The developed system can rank reports by the probability of being a *true positive* (i.e., a *real error*), thereby addressing the problem of a large number of false positives and making Infer a more practical tool because the current percentage of false positives is too high.

The report ranking system is based on *graph neural networks* (GNNs), which have become increasingly popular for code-related tasks in recent years because many *code properties* can be naturally expressed using graphs. A *dataset* is necessary to train GNN models. This thesis utilizes the D2A dataset [94], which contains *labeled* (true positive vs false positive) Infer reports from 6 open-source projects. D2A includes samples in a textual form, which must be converted into a graph form. For this reason, a training pipeline was created that generates Graph D2A – D2A transformed into a graph form. The samples in Graph D2A cannot be directly used for training *GNN models*; *feature engineering* must first be applied to them. Feature engineering optimizes the graphs and transforms them into the format proposed in this thesis – *Extended Code Property Graphs* (ECPGs), which enrich

existing *Code Property Graphs* (CPGs) commonly used for *vulnerability* detection in source code using GNNs. In particular, we enrich them by *Call Graphs*, *data types*, and other information.

The developed GNN models were trained using ECPGs from the training sets of 3 D2A projects, namely httpd, libtiff, and nginx. The models were evaluated on the test sets of the same projects. The experimental results show that using the models we obtain comparable, and in some cases even superior results than the existing *state-of-the-art* solutions, which are developed by strong industrial teams from IBM [94, 68]. These results demonstrate that the created models are a suitable open-source alternative to the compared existing solutions, all of which are – to the best of our knowledge – *closed source*.

The models were also tested using *cross-analysis* – a model is tested on a different project than it was trained on. The models proved insufficient for this challenge, highlighting the difficulty of cross-analysis in this area of research, as none of the existing compared solutions function in cross-analysis either.

The last contribution of this thesis is the inference pipeline, which can run Infer analysis, generate an ECPG for each report, and finally sort the reports using the created GNN models, for any C (and subset C++) software. This pipeline is based on the author's bachelor's thesis, which dealt with automating Meta Infer analysis. This pipeline, originally designed for cross-analysis, can also be used for inference on projects with sufficient history, on which the GNN models were trained.

Structure of the thesis The rest of the thesis is structured as follows. Chapter 2 explains the basic concepts of static analysis, Meta Infer, graph neural networks, graph representations used, and finally describes the tools used – LLVM Slicer, LLVM2CPG, and Joern. Chapter 3 describes the D2A dataset, its creation principle, comparison with other datasets, and the reasons for choosing D2A. Chapter 4 describes the design of the training pipeline, inference pipeline, and the proposed architecture of the GNN models. The implementation of the models and both pipelines is described in Chapter 5. The results of experiments and comparison with existing models are in Chapter 6. Finally, the conclusion is presented in Chapter 7. The thesis also includes Appendix A with the content of the attached media and of the additional resources available in the Zenodo trusted repository, Appendix B with installation instructions and user manual, and Appendix C with additional figures and tables.

Acknowledgement This thesis is a collaboration with Red Hat. It is also supported by the H2020 ECSEL Valu3s, GACR AIDE 23-06506S, and IGA FIT-S-23-8151 projects.

Chapter 2

Preliminaries

This chapter introduces the basic concepts, principles, and tools on which this thesis builds. Specifically, Section 2.1 briefly describes *static analysis*, its applications, advantages, and limitations. Section 2.2 describes the Meta Infer static analyzer, its use, types of *detectable errors*, and its advantages and disadvantages. Section 2.3 describes the general principle of *graph neural networks*, their advantages for *source code analysis*, and especially their input format. Section 2.4 introduces the different *source code representations* used as input to graph neural networks and focuses on the most commonly used type – *code property graphs*. Section 2.5 presents the LLVM-Slicer for *slicing* LLVM bitcode. Section 2.6 describes the LLVM2CPG tool for constructing code property graphs from LLVM bitcode. Finally, Section 2.7 presents the Joern platform used for various static analysis tasks.

2.1 Static Analysis

Static analysis [3, 23, 37] can be understood as a way of *reasoning* about the *run-time properties* of computer programs without the need to run them (at least not under their original semantics) or provide their inputs. Using static analysis, it is possible to investigate program properties such as *time* or *memory complexity*, look for errors such as *null pointer dereferences*, *accesses beyond array boundaries*, improper handling of resources, etc. It is also possible to check for *synchronization errors* such as *deadlocks*, *data races*, *atomicity violations*, etc. Finally, static analysis can be used to ensure compliance with language standards, e.g., MISRA-C/MISRA-C++[1] or compliance with practices for writing readable code, e.g., Google Java Style[2].

The opposite of static analysis is *dynamic analysis*, which requires running the program to be analyzed and thus a need to provide inputs. Since both approaches have their advantages and disadvantages, it is not advisable to use only one, but rather to use both simultaneously to complement each other. The advantages of static analysis are [3, 42]:

- Static analysis implicitly considers all possible paths in the code (even the rarely executing ones),

[1] **MISRA**'s website: https://www.misra.org.uk/.
[2] **Google Java Style Guide**: https://google.github.io/styleguide/javaguide.html.

- can report the exact location of the error and thus speed up the fix,
- does not require executable, sometimes even compilable source code, so errors can be detected early in the development,
- can be run fully automatically, after some initial setup.

However, static analysis also has its disadvantages [37, 42]:

- The initial setup can be tedious for some tools as it may require, e.g., creating *models* of certain functions, access to the compilation commands, or manually defining the required style guide.
- Running heavier-weight static analysis can be time and memory consuming.
- Static analyzers can report *false positives* (i.e., false errors) or *false negatives* (i.e., missed real errors).

The *Rice's theorem* implies [62] that all non-trivial properties of program behavior are *undecidable*. From this, it follows that in order to derive such properties automatically, it is necessary to introduce some degree of *approximations*. This approximation is the cause of false positives and false negatives. However, if a suitable approximation is used, it is possible to use static analysis to prove some properties (as opposed to dynamic analysis) – typically the absence of errors. An example of this behavior is the use of Frama-C to create an RTE-free[3] X.509 parser [22]. However, most tools try to create approximations that balance the number of false positives and false negatives to make the tools practical to use.

2.2 Meta Infer

Meta Infer [25] (formerly Facebook Infer) is an open-source[4] *framework* for writing *intraprocedural* and *interprocedural* static analyses [36, 58, 59]. Although it is a framework, Infer already includes a number of default and non-default (i.e., they must be explicitly enabled) analyses. Individual analyses are plugged into Infer in the form of *plugins*. Different plugins use different principles to detect different types of errors, e.g. InferBO, which uses the *symbolic interval* technique [46] to detect incorrect array indexing, or the Bi-abduction plugin, which uses *bi-abduction* [27] – a form of *inference* for *separation logic* that models computer memory – to detect errors associated with incorrect memory manipulation. Among other issues, Infer can detect null pointer dereferences, *dead stores*, *uninitialized values*, deadlocks, data races, *variable overflows*, and many other types of errors. Table 2.1 lists all the plugins that Infer provides, along with information about the language support and whether the plugin is enabled by default. More detailed information about each plugin and the types of errors reported by Infer can be found in [26].

Infer plugins are not *sound*, which, in the context of finding errors, means that they may have false negatives. Instead, Infer aims for maximal practical use – scaling to millions

[3] **Run Time Error (RTE)**.
[4] **Meta Infer**'s repository: https://github.com/facebook/infer/.

Table 2.1: Language support information for all non-experimental Infer plugins, along with whether the plugins are enabled by default.

Plugin	C	C++	Objective C	Java	C#	Default
Annotation Reachability	✓	✓	✓	✓	✓	
Bi-abduction	✓	✓	✓	✓	✓	✓
InferBO	✓	✓	✓	✓	✓	
Cost	✓	✓	✓	✓	✓	
Eradicate				✓	✓	
Impurity	✓	✓	✓	✓	✓	
Inefficient keySet Iterator				✓	✓	✓
Litho „Required Props“				✓	✓	
Liveness	✓	✓	✓			✓
Loop Hoisting	✓	✓	✓	✓	✓	
Pulse	✓	✓	✓	✓		
Purity	✓	✓	✓	✓	✓	
Quandary	✓	✓	✓	✓	✓	
RacerD		✓		✓	✓	✓
.NET Resource Leak					✓	✓
SIOF		✓				✓
Self in Block		✓	✓			✓
Starvation	✓	✓	✓	✓	✓	✓
Uninit	✓	✓	✓			✓

of lines of code thanks to *modular analysis*. It is also very simple to use [24] compared to other analyzers. Infer takes as an input *compilation commands* that allow the Infer's internal clang compiler to transform source files into the SIL[5] internal representation [4, 89]. This transformation (*capture*) of the source code takes place in the *capture phase*. To facilitate the capture of compilation commands, Infer supports a variety of *build systems* such as ant, cmake, Gradle, Make, Maven, and others. However, experiments conducted in previous work by the author [3] show that this support is incomplete and often fails to capture compilation commands. Therefore, as part of the same work, a compiler wrapper was created that can reliably capture compilation commands and pass them to Infer.

The capture phase is followed by an *analysis phase* in which the required plugins are run over the SIL. The output of Infer after the analysis phase is a list of found errors. Experiments on *real-world* programs in previous work [3] also show that Infer has a very high number of false positives. Specific numbers suggest approximately 4.5 false positives for every real error. However, this score is very optimistic since it includes dead store errors, which are harmless and can be detected by common compilers and are present in real-world programs in very large numbers, especially in the C language when using conditional compilation. Without dead stores, the number increases to approximately 9 false positives for every real error. In general, such a high number of false positives in static analyzers results in developers' distrust of these tools and consequent ignoring of analysis results [18, 44, 63]. Therefore, efforts are made to reduce false positives.

[5] **Smallfoot Intermediate Language (SIL)**.

2.3 Graph Neural Networks

There are a number of approaches for detecting errors in programs using *machine learning* [35]. The approaches can be divided into *convolutional neural networks* (CNNs) [19], *recurrent neural networks* (RNNs) [49, 50, 51, 52, 71, 96], and graph neural networks (GNNs) [9, 13, 30, 73, 75, 95], depending on the *architecture* of the model used. These approaches are often combined with each other [28, 47, 69, 70].

CNNs achieve very good results, e.g., in *image classification*. This is aided by *convolutional layers* that can appropriately capture *spatial information* from an image. However, this principle is not so effective for source code [60]. In order to use the source code as input to a CNN, it must first be transformed into a graph and then into a matrix (e.g. an *adjacency matrix*). Due to the fact that the nodes in a graph do not have a fixed order, the same graph can be expressed as an adjacency matrix (which has a fixed order of nodes) in multiple ways. This property is very undesirable because a single result is wanted for the same graph. It also makes it impossible to use the *local spatial properties* of convolutional layers. Another problem for CNN, is the arbitrary size of the graph since the adjacency matrix have a fixed size.

Another frequently used approach is to represent code as a *sequence*, especially for recurrent neural networks. This approach is based on the idea that the source code can be treated as a *natural language*. While these approaches achieve very good results [8, 34, 68], the properties of source code can be better represented using graphs. Appropriately designed graphs can more explicitly model properties between parts of the code that would otherwise the model had to learn during training. The idea that a graph is a better representation of source code than aa adjacency matrix or a sequence is supported by the experiments in [75], especially on *synthetic datasets* (on *datasets* with real-world examples, all approaches seem to perform poorly). Arbitrary input sizes can also pose problems for RNNs, as they may have a limited input sequence length [68].

GNNs are designed to work on arbitrarily large graphs. For this reason, and the previously mentioned reasons, GNNs were chosen for this thesis. Therefore, a brief description of a general graph neural network based on *message passing* follows. The description uses a slightly modified notation from [48].

Consider an *oriented graph* structure $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the *set* of *nodes* and $\mathcal{E}$ is the set of *oriented edges* $e = (v, v') \in \mathcal{V} \times \mathcal{V}$. The source node of an edge $e = (v, v')$ is v and the target node is v'. The *embedding vector* of a node v is denoted by $\mathbf{h}_v \in \mathbb{R}^D$ where D is the *dimension* of the vector. Each node has a *label* which is denoted by $l_v \in \{1, ..., L_\mathcal{V}\}$, and each edge has a label which is denoted by $l_e \in \{1, ..., L_\mathcal{E}\}$. Further, auxiliary sets of nodes are defined. The set $\textsc{In}(v) = \{v' | (v', v) \in \mathcal{E}\}$ contains the *predecessors* of a node v. The set $\textsc{Out}(v) = \{v' | (v, v') \in \mathcal{E}\}$ contains the *descendants* of a node v. *Bi-directional propagation* then proceeds by updating each node until *convergence* (or for a fixed number of steps) using the following formula:

$$\mathbf{h}_v^{(t)} = \sum_{v' \in \textsc{In}(v)} f(l_v, l_{(v',v)}, l_{v'}, \mathbf{h}_{v'}^{(t-1)}) + \sum_{v' \in \textsc{Out}(v)} f(l_v, l_{(v,v')}, l_{v'}, \mathbf{h}_{v'}^{(t-1)})$$

Here, the function f can be a *linear function* or a *neural network*. For each node v, the output of this network is defined as $o_v = g(\mathbf{h}_v^{(T)}, l_v)$ where g is an arbitrary *differentiable*

function and T is the final iteration. In case where *graph-level classification/regression* is needed, it is possible to artificially add a so-called „*super node*“ to the original graph, which will be connected to all nodes. This will allow graph-level classification/regression to be treated in the same way as *node-level classification/regression*.

The above description of how the information is propagated in GNNs shows that the graphs used as inputs, must form a single WCC[6] in the case of bi-directional GNNs. And for *directional GNNs* the edges must also be properly oriented (more information in Section 4.1.4). If the graph does not meet these properties, it is not possible to pass information between WCCs within *GNN updates* (this needs not be a problem for some types of tasks, but it is crucial for the system designed in this thesis). If the function f is *differentiable*, then all components are differentiable, and after T iterations, it is possible to compute gradients of the parameters (typically located within the function f) and train the GNN layers using *gradient descent*.

2.4 Source Code as a Graph

There are many types of graphs that are commonly used as source code representations, e.g., *abstract syntax trees* (AST), *control flow graphs* (CFG), *program dependency graphs* (PDG), and others. One type of such commonly use graphs is the *code property graph* (CPG), which is composed of all three previously mentioned graphs and used in its pure form, e.g., in [53, 75]. Modified versions of it are often used as well, e.g., *simplified CPGs* (SCPG) for *function-level vulnerability detection*[7] in C/C++ [90], CPGs with added edges that reflect the original order of *tokens* (i.e., individual source code elements) [95], or *code composite graphs* (CCG) again for vulnerability detection in C/C++ [9]. Furthermore, PDGs alone are used, e.g., for finding *malicious code* in JavaScript [28], XFGs (*subgraphs* of PDGs) for detecting vulnerabilities in C/C++ code [13]. Or CFGs together with token sequences for detecting vulnerabilities in PHP [69].

According to [92], the reason for the creation of CPGs is the inability of each subgraph type to detect certain types of errors independently during *traversal*. For example, ASTs are not suitable for detecting *divisions by zero*. However, by combining ASTs and PDGs, this is possible, but one still cannot detect, e.g., integer overflows. This can only be detected by combining ASTs, CFGs and PDGs. This combination of graphs results in a representation that is able to capture both syntactic and semantic properties of the code and preserve most types of errors in it. Exceptions are, e.g., *race conditions*, which need more external information. A complete table of detectable errors and required graph types is given in [92]. The following graph definitions are based on the original definitions from the paper introducing CPGs [92], with only minor changes in notation to resemble the GNN definition given earlier. It should be noted that the following definitions employ an abuse of notation, as it was used in the original paper.

To define a CPG, it is first necessary to define a *property graph* [92], which is a commonly used graph type in *graph databases* such as Neo4j. A property graph is an oriented *multigraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \lambda, \mu)$, where $\mathcal{V}$ denotes the set of nodes, $\mathcal{E}$ denotes the set of edges

[6] A **Weakly Connected Component (WCC)** is a set of nodes where there is a path between any two nodes, without respecting the direction of the edges.

[7] In some articles, the terms ”error” and ”vulnerability” are used interchangeably, but not every error is a vulnerability.

```
1 void foo()
2 {
3     int x = source();
4     if (x < MAX)
5     {
6         int y = 2 * x;
7         sink(y);
8     }
9 }
```

Listing 2.1: A code sample. The code was taken from [92].

$e = (v, v')$, where $v, v' \in \mathcal{V}$. λ denotes the *edge labeling function* $\lambda : \mathcal{E} \to \Sigma$, with $\Sigma = 1, ..., L_{\mathcal{E}}$ being the edge labels. Finally, μ denotes the function $\mu : (\mathcal{V} \cup \mathcal{E}) \times K \to S$ that assigns attributes to nodes and edges where K is the set of *attribute* names and S is the set of attribute values.

An AST [92] is an *ordered tree* whose inner nodes represent *operators* and outer nodes (*leaves*) represent *operands*. The oriented edges then show the parenting relation. The AST captures the syntactic nature of the code. Consider the code in Listing 2.1. The AST constructed for this code is shown in Figure 2.1.

To create a CPG definition, the subgraph types must be converted to the same format – in particular, to the previously defined property graphs. An AST as a property graph is a structure $\mathcal{G}_{\text{AST}} = (\mathcal{V}_{\text{AST}}, \mathcal{E}_{\text{AST}}, \lambda_{\text{AST}}, \mu_{\text{AST}})$, where $\mathcal{V}_{\text{AST}}$ is the set of AST nodes and $\mathcal{E}_{\text{AST}}$ is the set of AST edges. The function λ_{AST} is defined as $\lambda_{\text{AST}}(v) =$ 'AST' and is applied to each node $v \in \mathcal{V}_{\text{AST}}$. The function $\mu_{\text{AST}} : \mathcal{V}_{\text{AST}} \times K_{\text{AST}} \to S_{\text{AST}}$ is applied to each node and attribute. The attribute names are $K_{\text{AST}} = \{$'*code*', '*order*'$\}$ and the attribute values are $S_{\text{AST}} = S_{code} \cup S_{order}$, where S_{code} are types of nodes in an AST, e.g., *variable*, *constant*, mathematical operators, etc., and S_{order} assigns values that order a node among its siblings in the AST to preserve the ordering from the original tree.

A CFG [92] is an oriented graph describing the possible paths of *program control* and the conditions for their *execution*. The nodes of the graph represent *statements* and *predicates*, while the edges represent *control passing*. Each command node has an outgoing edge labeled ε, which denotes an *unconditional* passing of control. While a predicate node must have two outgoing edges *true* and $false$ for different evaluations of a given predicate. Consider the code in Listing 2.1. The CFG constructed for this code is shown in Figure 2.2. The CFG as a property graph is the structure $\mathcal{G}_{\text{CFG}} = (\mathcal{V}_{\text{CFG}}, \mathcal{E}_{\text{CFG}}, \lambda_{\text{CFG}}, \cdot)$ where $\mathcal{V}_{\text{CFG}}$ is the set of nodes corresponding to the nodes from the AST as follows:

$$\mathcal{V}_{\text{CFG}} = \{v \in \mathcal{V}_{\text{AST}} \mid \mu_{\text{AST}}(v, \text{'}code\text{'}) \in \{\text{'STMT'}, \text{'PRED'}\}\}$$

The edge labeling function is defined as $\lambda_{\text{CFG}} : \mathcal{E}_{\text{CFG}} \to \Sigma_{\text{CFG}}$ where the values in the set $\Sigma_{\text{CFG}} = \{$'*true*', '$false$', 'ε'$\}$ correspond to the meaning of edges in the CFG.

A PDG [92] is again an oriented graph whose nodes are statements and predicates. There are two types of edges in a PDG, namely *data dependency edges*, which model the influence of a variable on the value of another variable, and *control dependency edges*, which

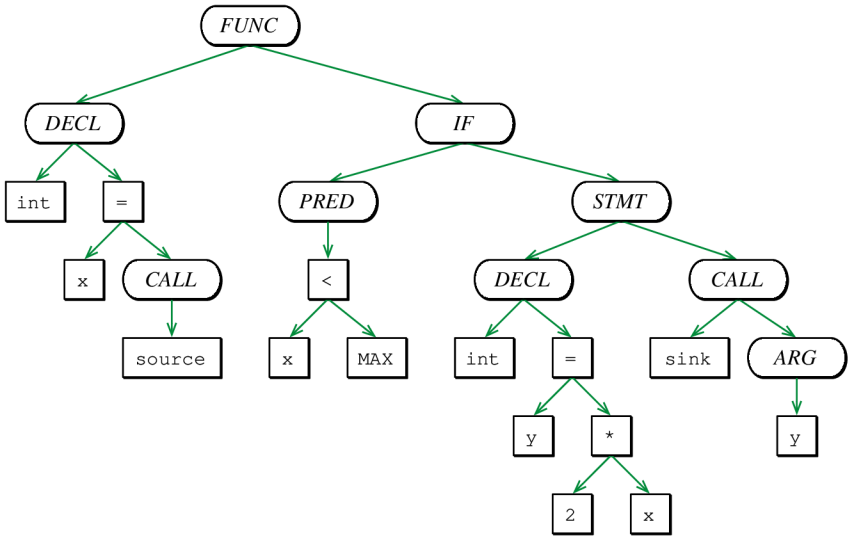

Figure 2.1: The abstract syntax tree for the code in Listing 2.1. This figure was taken from [92].

model the influence of predicates on the values of variables. Consider the code in Listing 2.1. A PDG constructed for this code is shown in Figure 2.2. The PDG as a property graph is a structure $\mathcal{G}_{\text{PDG}} = (\mathcal{V}_{\text{CFG}}, \mathcal{E}_{\text{PDG}}, \lambda_{\text{PDG}}, \mu_{\text{PDG}})$ where the nodes are the same as in the CFG. The edge labeling function is defined as $\lambda_{\text{PDG}} : \mathcal{E}_{\text{PDG}} \to \Sigma_{\text{PDG}}$, where the edge labels $\Sigma_{\text{PDG}} = \{\text{'}data\text{'}, \text{'}control\text{'}\}$ correspond to the meaning of edges in the PDG. The function assigning attribute values has the form of $\mu_{\text{PDG}} : \mathcal{E}_{\text{PDG}} \times K_{\text{PDG}} \to S_{\text{PDG}}$, where $K_{\text{PDG}} = \{\text{'}symbol\text{'}, \text{'}condition\text{'}\}$ and $S_{\text{PDG}} = S_{\text{VAR}} \cup \{\text{'}true\text{'}, \text{'}false\text{'}\}$. The set S_{VAR} represents the set of names of all variables that occur as the output node of the data dependency edges. The function μ_{PDG} then works by assigning the value of the attribute '*symbol*' to the '*data*' edges as the name of the variable represented by the source node of the edge, and '*control*' edges are assigned the attribute value '*condition*' depending on whether they are in the *true* or *false* branch.

The CPG is then defined using the previous definitions of AST, CFG, and PDG as:

$$\mathcal{G} = (\mathcal{V}_{\text{AST}}, \mathcal{E}_{\text{AST}} \cup \mathcal{E}_{\text{CFG}} \cup \mathcal{E}_{\text{PDG}}, \lambda, \mu)$$

where the definition of the function λ is as follows:

$$\lambda(e) = \begin{cases} \lambda_{\text{AST}}(e) & \textit{if } e \in \mathcal{E}_{\text{AST}} \\ \lambda_{\text{CFG}}(e) & \textit{if } e \in \mathcal{E}_{\text{CFG}} \\ \lambda_{\text{PDG}}(e) & \textit{if } e \in \mathcal{E}_{\text{PDG}} \end{cases}$$

and the definition of the μ function is:

$$\mu(x, p) = \begin{cases} \mu_{\text{AST}}(x, p) & \textit{if } (x, p) \in \mathcal{V}_{\text{AST}} \times K_{\text{AST}} \\ \mu_{\text{PDG}}(x, p) & \textit{if } (x, p) \in \mathcal{E}_{\text{PDG}} \times K_{\text{PDG}} \end{cases}$$

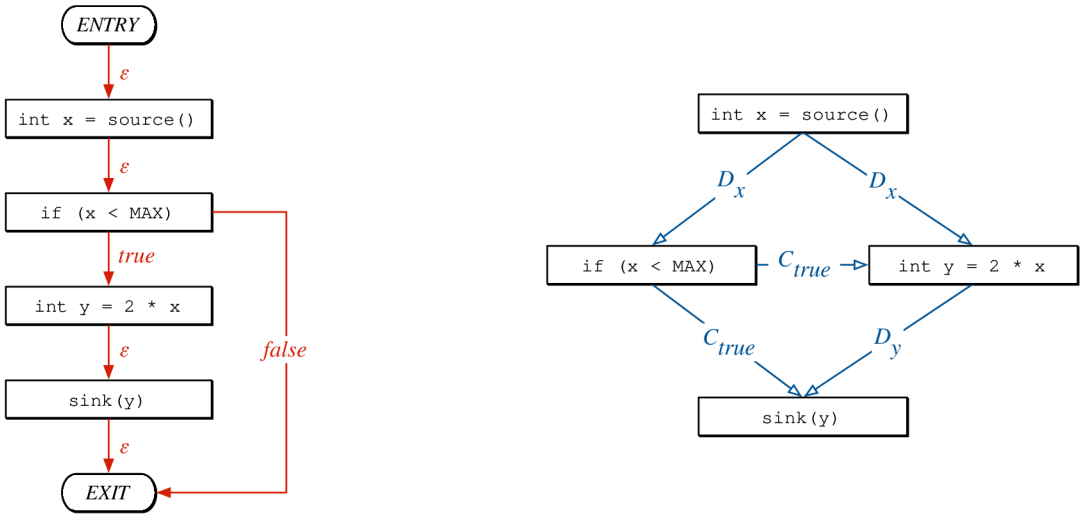

Figure 2.2: The control flow graph (on the left) and the program dependence graph (on the right) for the code in Listing 2.1. These figures were taken from [92].

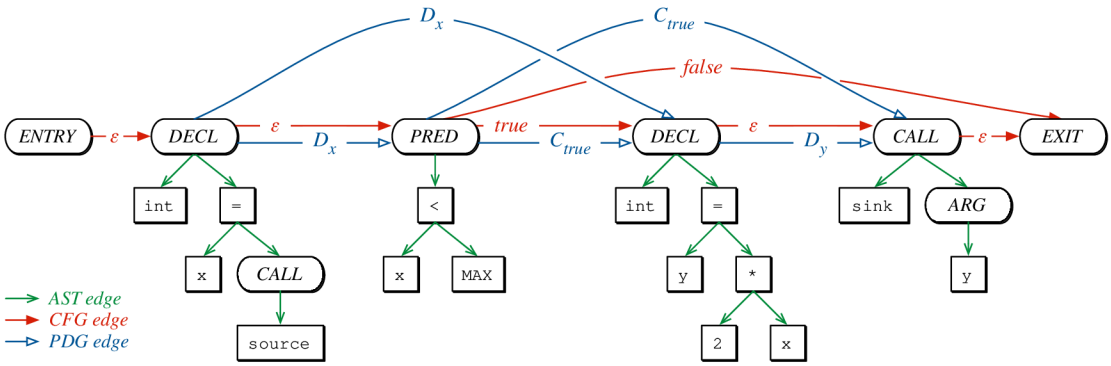

Figure 2.3: The code property graph for the code in Listing 2.1. This figure was taken from [92].

A CPG for the code in Listing 2.1 is shown in Figure 2.3 where the irrelevant `FUNC`, `IF`, and `STMT` nodes were omitted for demonstration purposes. And also an entry point and an exit point were added.

2.5 LLVM-Slicer

As described in more detail in Section 4.1.2, the LLVM-slicer is used for program slicing in this thesis. LLVM-slicer is an open-source[8] tool which uses the DG library [10, 11]. The DG library implements various interprocedural static analyses – namely, *pointer analysis*, *data dependence analysis*, *control dependence analysis*, and *value relationship analysis*. These analyses are implemented in DG as independent of the input language. However, the front-end currently supports LLVM bitcode only [54]. LLVM bitcode is a *storage format* for

[8] **LLVM-Slicer**' repository: https://github.com/mchalupa/dg.

LLVM IR[9] [55], which is an *assembly language* used as a low-level representation of code during the various stages of LLVM compilation.

The main use of the DG library is the aforementioned LLVM-slicer, which uses the DG analyses for *program slicing* – removing pieces of code that have no effect on *user-defined* areas in the code. Results of experiments with LLVM-Slicer on benchmarks from the Software Verification Competition can be found in [11]. Although LLVM bitcode is language-independent and can be generated from, e.g., C, C++, or Rust, LLVM-slicer does not support certain constructs in LLVM bitcode that handle *exceptions*. This means that it is not able to handle a C++ program that uses exceptions. If the C++ code is exception-free, it should be able to slice it. The input to the LLVM-slicer is a **single LLVM bitcode** file and slicing criteria. The output is the sliced LLVM bitcode.

Slicing criteria are specified, for example, using the option `-sc`. This option allows for a relatively extensive specification of slicing criteria [12]. However, in this thesis, the basic format `-sc file#function#line#obj` is used only. The fields `file`, `function`, `line`, or `obj` can be empty. The meanings of `file`, `function`, and `line` are straightforward – they refer to locations in the code. The `obj` field maps to a function call or a variable use at the location (the code must be compiled with debugging information, see Section 4.1.1).

Furthermore, it is necessary to define an *entry point function* that must be present in the input bitcode. The default entry point is the `main` function. However, it can be overridden using the option `-entry=function`. The entry function acts as the starting point for the analysis – anything above this function in the call tree is removed.

2.6 LLVM2CPG

In this thesis, the open-source[10] LLVM2CPG tool is used for generating CPGs from LLVM bitcode, as detailed in Section 4.1.2. The CPGs were originally created for high-level languages such as C, which creates some problems when creating CPGs from low-level LLVM IR [16]. One problem is mapping LLVM IR instructions to classical high-level operations in order to display the CPG in the same format as, e.g., for the C source code. Some operations can be mapped directly because they have the same *semantics*, others can be modeled using functions, and some cannot be mapped at all and need to be bypassed by another mechanism. The CPG output format can be further processed by Ocular[11] (proprietary), Plume[12] (open-source) or Joern (open-source, see Section 2.7).

2.7 Joern

Joern is used in this thesis to enrich CPGs with additional information, as detailed in Section 4.1.2. Joern [87] is a powerful open-source[13] platform providing various tools from the area of static analysis. Using Joern, it is possible to write custom static analyses

[9] **LLVM Intermediate Representation (LLVM IR)**.
[10] **LLVM2CPG**'s repository: https://github.com/ShiftLeftSecurity/llvm2cpg.
[11] **Ocular**'s documentation: https://docs.shiftleft.io/ocular/quickstart.
[12] **Plume**'s documentation: https://plume-oss.github.io/plume-docs/.
[13] **Joern**'s repository: https://github.com/joernio/joern.

or *queries* over source files. Joern supports various *programming languages*, such as C, C++, JavaScript, Kotlin, Python, or Java. It is also possible to construct different graph representations of the code (ASTs, CFGs, CDGs, DDGs, PDGs or CPGs), which can be exported in different formats, e.g., DOT [32] or csv for the Neo4j graph database [65]. It is also possible to load already constructed CPGs in different formats, e.g., in the output format of the LLVM2CPG tool. Joern can be used as a *command line* tool, through an *interactive environment*, or as an *integration library*.

Chapter 3

D2A Dataset

This chapter introduces the D2A dataset, which is used in this work to train a system that reduces false positives of the Meta Infer static analyzer. Specifically, the chapter discusses the creation of D2A, the structure of individual samples, comparisons with other existing datasets, and presents statistics regarding the distribution of Meta Infer's error types. This chapter draws primarily from [94, 40].

D2A is a dataset developed by IBM, containing errors found by the Meta Infer static analyzer and information about their validity (true positive/false positive). D2A was first introduced in [94] and is freely available for download at [40]. The dataset is generated automatically based on *differential static analysis*, and the source files for the D2A generation pipeline are open-source[1]. The dataset fits into the area of static analysis and is primarily intended for creating models aimed at eliminating false reports produced by static analyzers. Initial results from models such as Catboost, LightGBM, Random Forest, Extra-Trees, or the voting model can already be found in the article introducing D2A [94]. The team behind D2A also later published the work [68], where they improve the existing models and add the C-BERT model, which is *Bidirectional Encoder Representations from Transformers* [17], but trained on C code and *fine-tuned* on the D2A dataset for the purpose of classifying reports.

Several reasons led to the selection of the D2A dataset for this thesis:

1. It is created from real-world open-source projects.
2. Meta Infer was used for differential static analysis (thus, samples contain trace, location, error type, etc., which is necessary for extracting slicing information, more information in Section 4.1.2).
3. Being an automatically generated dataset, it is sufficiently large.
4. The author of this thesis has previously collaborated with the creators of the D2A dataset.

[1] **D2A pipeline**'s repository: https://github.com/IBM/D2A.

Figure 3.1: A schematic of the D2A generation pipeline. This figure was taken from [94].

Dataset Creation Principle

The D2A dataset was automatically generated using differential static analysis on open-source projects with extensive git histories. The schematic of the pipeline for generating D2A is shown in Figure 3.1. The fundamental concept of this pipeline is that the git history includes commits that fix real errors. Therefore, the entire pipeline starts with the identification of these potential fixing commits. These commits are identified using the Commit Message Analyzer, which, through similarity-based methods and key phrase search in commit messages, can select commits that are highly likely fixing errors. For each such commit, Meta Infer (see Section 2.2) is run on the version of the code before and after the commit. Errors that are found in the before version and are missing in the after version are considered true positives, indicating they have been fixed. For an error to be counted as a true positive, it must also satisfy the following conditions:

1. The error must not appear in later versions.
2. The commit must have modified some part of the bug trace[2] related to the error.

All other errors are considered false positives – this is, of course, an approximation because otherwise it would imply that the project in its latest version contains no errors, which is highly unlikely. The D2A dataset also includes another type of sample called *after-fix* samples, which are labeled as false positives. Each after-fix sample is generated as a counterpart to a true positive sample on the after version, where the corresponding true positive have been fixed – the after-fix samples contain the fixed code. After-fix samples have the property of creating a balanced dataset along with the true positives and also form pairs that can help models learn to differentiate between true positives and false positives. This is because the pairs provide the models access to the same code with and without the error. However, these samples naturally do not have Meta Infer outputs and are not used in this thesis.

As previously mentioned, each sample includes the output from Meta Infer, the code of the functions related to the error, and additional metadata such as the ID, label, commit hash, and compiler arguments for all files affected by the error (this is possible because Meta Infer needs to compile the code as discussed in Section 2.2). The complete list of sample attributes is too extensive to be included here, but it is documented in [39]. Attributes and their formats necessary for the further explanation will be described in later chapters.

[2] A **bug trace** is information attached to some outputs of Meta Infer. It includes sections of the code that influenced the particular error.

Comparison with Other Existing Datasets

There are numerous datasets designed for training models that identify errors in C/C++ code as can be seen in the table comparing existing datasets with D2A in [94]. These datasets are typically categorized into synthetic and real-world types. Synthetic datasets offer the advantage of 100 % label accuracy and the ability to automatically generate samples, making them sufficiently large. However, synthetic samples are typically simpler and differ from real code, which may lead to poor *generalization* when applied to real-world software. Real-world datasets can be further divided into manually and automatically created. Manually created datasets are highly accurate but are typically too small. Automatically created datasets, on the other hand, suffer from lower accuracy but are large enough. The D2A dataset employs a hybrid approach, automatically generating samples from real-world projects while striving to identify bug-fixing commits that were manually corrected. As a result, the dataset achieves an accuracy where the true positive class has accuracy of 41 % and false positive class has accuracy of 81 %. These accuracies were determined through manual validation of 41 samples labeled as true positive and 16 samples labeled as false positive.

Dataset Distribution

The D2A dataset includes 6 open-source projects–openssl[3], libav[4], nginx[5], libtiff[6], httpd[7], and FFmpeg[8]. While it is theoretically possible to expand it to include any software with a sufficient history of commits, generating it is computationally demanding as it requires running Meta Infer twice for each targeted commit on the entire project. The D2A dataset contains a total of 1,314,276 samples and is provided with a split into *training*, *validation*, and *testing datasets* to match the results of the models from [94, 68]. Each sample is labeled either true positive (1) or false positive (0) and categorized by error type as determined by Meta Infer outputs, such as `NULL_DEREFERENCE`, `UNINITIALIZED_VALUE`, etc. Tables C.1 and C.2 show the counts of samples according to the label, project, and error type. The tables also highlight the types of errors supported by the system for reducing false positives in this thesis, with more details available in Section 4.1.1.

[3] **openssl**'s repository: https://github.com/openssl/openssl.
[4] **libav**'s repository: https://github.com/libav/libav.
[5] **nginx**'s repository: https://github.com/nginx/nginx.
[6] **libtiff**'s repository: https://gitlab.com/libtiff/libtiff.
[7] **httpd**'s repository: https://github.com/apache/httpd.
[8] **FFmpeg**'s repository: https://github.com/FFmpeg/FFmpeg.

Chapter 4

Design of a System for Reducing False Positives in Meta Infer

This chapter describes the design of training and inference pipelines for transforming the D2A dataset into its graphical form, referred to as Graph D2A. Specifically, Section 4.1 describes the training pipeline, which transforms the D2A dataset into Graph D2A. Section 4.1.1 focuses on the bitcode generation phase, aiming to produce LLVM bitcode from the D2A dataset samples. Section 4.1.2 explains the creation of extended code property graphs from the generated LLVM bitcode. Section 4.1.3 provides a detailed description of the Graph D2A format. Section 4.1.4 discusses the feature engineering process, which converts graphs from Graph D2A into an optimized input format for graph neural networks. Section 4.1.5 outlines how to train graph neural networks using these optimized graphs.

Section 4.2 addresses the design of the inference pipeline, a modification of the training pipeline designed to automatically extract graphs and apply the graph neural network models to any real-world C (and a subset of C++) software. Specifically, Section 4.2.1 describes the capture phase, aimed at running Meta Infer analysis and extracting LLVM bitcode from the build of real-world software. Finally, Section 4.2.2 discusses the inference phase, which deploys the trained models on the created graphs and ranks a list of errors detected by Infer based on the likelihood of being true positives.

We recall that he goal of this thesis is to create a system to reduce false positives from the static analyzer Meta Infer, described in Section 2.2. Due to reasons mentioned in Section 2.3, graph neural networks (GNNs) were chosen for this task. The goal of the trained models is to rank the errors found by Infer based on their likelihood of being true positives. The D2A dataset was selected for reasons detailed in Chapter 3. Although D2A includes the source code of functions mentioned in Infer's bug traces, this information is stored as text (more specifically as JSON) and not as graphs. To enable the training of GNNs on D2A, it first needs to be transformed into an appropriate graph format. According to Section 2.4, a suitable and frequently used representation are the code property graphs (CPGs) and its modified versions. The application of GNNs to source code requires a preliminary mechanism for graph construction. However, the existing graph construction methods have several limitations, which led to the development of our training and inference pipelines. The three main disadvantages of the current solutions are:

1. Insufficient graph representations, such as constructing only ASTs [73], XFGs [13], or CFGs [69][1].

2. Not considering conditional compilation [9, 33, 75, 91, 95].

3. The inability to automatically construct graphs for any software [9, 33, 75, 91, 95].

Points 2) and 3) are closely linked. The previous works, namely, [9, 33, 75, 91, 95], all use the Joern tool (see Section 2.7) to construct CPGs (and its various modifications), that is why they are mentioned in both 2) and 3). Although Joern is a very useful tool, its disadvantage is that it analyses the source files directly and is not able to connect to the build process itself. This makes it unable to identify which source files to process and which not to. While Joern can *recursively* find and process source files in a given directory [88], it does indeed process everything it finds in those directories. This becomes a problem if the software includes different versions of the source code, e.g. for different *operating systems* (Windows or Linux), which are selected only during compilation. Joern will thus not be able to correctly construct a CPG without knowing which file to use in a given context. Therefore, Joern cannot be fully automatically deployed on arbitrary software.

There is a similar problem with conditional compilation where Joern does not know which part of the code to use, or what values the macros have, since they can be (and very often are) defined during compilation. For this reason, Joern considers all macros as undefined by default, and therefore irretrievably loses code fragments that did not satisfy the conditions within `#ifdef` or `#ifndef` during preprocessing. These problems do not seem to manifest themselves in artificial datasets, and for concrete real-world software, these problems must be solved manually if using pure Joern.

Points 2) and 3) are also closely related to Infer – since its inputs are compilation commands, and the source code is compiled using them before the analysis (see Section 2.2) – Infer analyzes the preprocessed code. This means that the Infer's analysis is platform-dependent – it can find different errors under various compilation conditions. Therefore, it makes sense to construct graphs from the code as seen by Infer.

The use case of the proposed pipelines differs subtly from previous studies. In particular, we need to construct graphs based on the code in alignment with the Infer report that needs to be sorted. This requires the capability to slice the code according to the information extracted from the report. Program slicing is also employed in some earlier studies. However, in this regard, the most comparable studies, specifically [94, 68], do not use program slicing.

The proposed pipelines are intended to create extended CPGs (further discussed in Sections 4.1.3 and 4.1.4) from software written in C and a subset of C++. The limitation for C++ arises from the use of LLVM-Slicer, with specific reasons elaborated in Section 2.5.

4.1 Training Pipeline

The goal of the training pipeline is to transform the D2A dataset into its graph version – Graph D2A, upon which a GNN model will be trained. Figure 4.1 shows that the

[1] This work, however, employs a hybrid approach using both GNN and RNN.

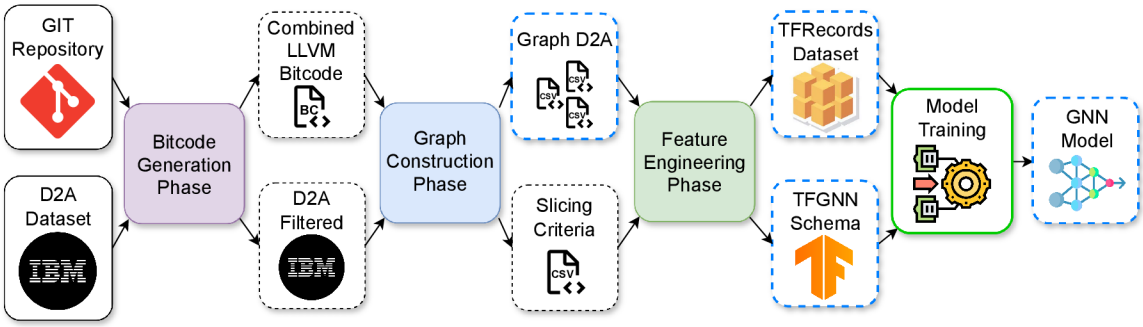

Figure 4.1: The figure shows a schematic of the training pipeline that transforms the D2A dataset into Graph D2A, and then trains models on it. Dashed boxes represent the intermediate products and data generated by the pipeline. A blue outline highlights the important outputs of the pipeline, and a green outline indicates the tool developed in this thesis (in addition to assembling and controlling the entire pipeline). The training pipeline includes phases such as bitcode generation, graph construction, and feature engineering, detailed in Sections 4.1.1, 4.1.2, and 4.1.4, respectively. Icons were taken from [103, 100, 66, 97, 104].

training pipeline consists of three stages – bitcode generation, graph construction, and feature engineering, each detailed in Sections 4.1.1, 4.1.2, and 4.1.4, respectively. The input to the entire pipeline is the D2A dataset along with the project repositories from which D2A was generated. The outputs of the pipeline:

1. For each project: a **Graph D2A dataset** – the D2A dataset transformed into raw extended code property graphs (ECPGs) in the CSV format (see Sectio 4.1.3), which can be used for training GNNs (not only for ranking static analysis reports).
2. For each project: a **Graph D2A dataset with feature engineering** (see Section 4.1.4) prepared in the commonly used TFRecords format (again, see Section 4.1.4) for GNN training.
3. Same for all projects: the **TFGNN schema** describing the format of the Graph D2A with feature engineering (see Section 4.1.4).
4. Might be same for all projects (see Chapter 6.2): the **GNN model** for ranking Infer reports.

Both the training and inference pipelines internally use a conversion to LLVM IR. Since many languages can be compiled into LLVM IR (see Section 2.5), the Graph D2A can, to some extent, be considered language-independent; consequently, the models trained on it can also be considered as such. However, it is still important to remember that the original language was C. There are several advantages to generating graphs from LLVM IR:

- The output graphs have a simpler structure (as LLVM IR is a much simpler language compared to, for example, C or C++).
- The existing tools like LLVM-Slicer and LLVM2CPG can be utilized.

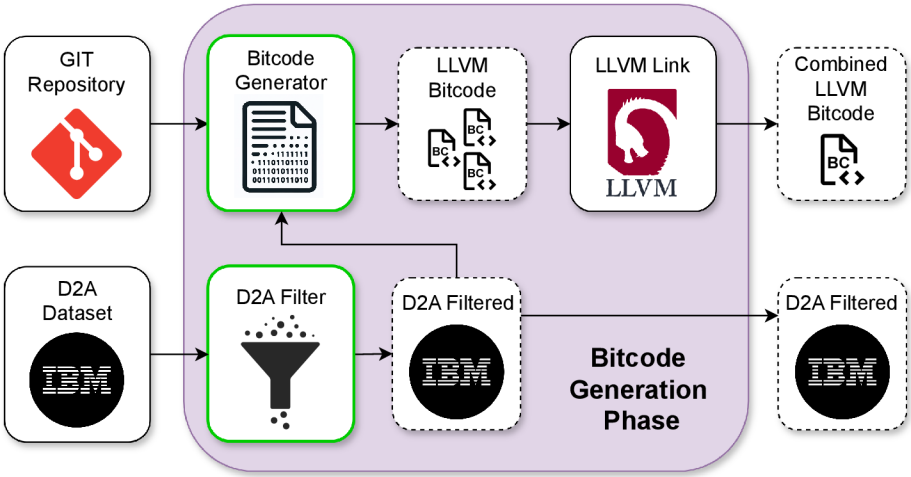

Figure 4.2: The figure shows a schematic of the bitcode generation phase, which generates LLVM bitcode for each D2A sample whose error type is supported. Dashed boxes represent the intermediate products and generated data. A green outline highlights the tools developed in this thesis. Icons were taken from [103, 100, 102].

- The output graphs are language-independent.

However, there are also disadvantages:

- The output graphs are larger in terms of the number of nodes and edges.
- It is not possible to transform the dataset directly; instead, a recompilation of individual D2A samples is necessary.

4.1.1 Bitcode Generation

LLVM bitcode is a binary representation of LLVM IR and can be freely converted between the two [54]. For conversion from LLVM IR to LLVM bitcode, the tool `llvm-as` (llvm assembler) is used, and for the reverse conversion, `llvm-dis` (llvm disassembler) is employed. However, these tools are not strictly necessary because, as shown in Figure 4.2, the bitcode generator directly produces LLVM bitcode, and all other parts of the pipeline (that work with LLVM IR) operate directly on LLVM bitcode as well.

The objective of the bitcode generator is to produce a set of LLVM bitcode files for each sample from the D2A dataset. The number of LLVM bitcode files for each sample is equal to the number of source files for that sample in D2A. The names of the source files for each sample can be extracted from the `compiler_args` attribute in D2A. Each sub-attribute in `compiler_args` follows the format [39]:

```
'file.c' : '-compiler\_arg1 -compiler\_arg2 ...'
```

Taking only the keys will produce a set of files (for a given sample) that need LLVM bitcode generation.

Before generating LLVM bitcode, it is essential first to filter the dataset and remove samples that will not be transformed. Tables C.1 and C.2 show that some error types have very few true positives. A small number of positive samples can make it difficult for models to train as they may not have sufficient information to learn the underlying patterns of true positives for those error types. Consequently, all error types with fewer than **200 true positives** across the entire dataset will be filtered out. `DEAD_STORE` errors are always true positives, as established in previous work by the author [3] and also confirmed by the D2A authors' experiments [94] who do not include `DEAD_STORE` errors in the manual verification of D2A.

An exception is made for the error types `BUFFER_OVERRUN_L1` and `INTEGER_OVERFLOW_L1`, which are included despite having only 28 and 22 true positives, respectively. The reason is that `BUFFER_OVERRUN_L1` is the same as, for example, `IBUFFER_OVERRUN_L5` – the only difference being that Infer is more certain of the truthfulness of `L1` than `L5` (more information in [3]), similarly for `INTEGER_OVERFLOW_L1`. Thus, both errors share the same underlying pattern, and the model should be capable of learning it. In the end, only errors with fewer than 30 true positives are removed, and even out of those, not all are removed, hence the overall data loss is minimal. However, this filtering implies a limitation that the model can only be applied to supported bug types which are highlighted in Tables C.1 and C.2.

Generating LLVM bitcode can be accomplished during the compilation using the `clang` compiler (all projects in D2A are written in C language) of the specified source file by inserting the following options [86] (which were recommended by the old version of Joern documentation[2], which unfortunately is no longer available) into the compilation command:

1. `-emit-llvm` – ensures that LLVM bitcode is used for *object files*.
2. `-g` – adds debug information which allows *backward mapping* of LLVM bitcode to the original source code, enabling the use of program slicing based on location information [12].
3. `-grecord-command-line` – inserts more debug information into the LLVM bitcode.
4. `-fno-inline-functions` – disables the use of *inline functions*.
5. `-fno-builtin` – prevents the compiler from inserting *built-in functions*.

The compilation command must indeed be specifically for compiling (it must include `-c`), and not for linking, preprocessing, etc. Additionally, the `-o` option along with its value must be removed, so that the compilation command generates a `.bc` (LLVM bitcode) file instead of the original file. The `compiler_args` attribute contains only options – typically just `-I` (include directories) and `-D` (definitions of macros and their values). Since neither `-c`, `-o`, nor the specific compiler used are mentioned among the options, it is unnecessary to remove `-o` or check if it is indeed a compilation command. Instead of the originally used compiler (which cannot be identified from D2A alone), `clang` will be used. Given that Infer also internally uses `clang` (see Section 2.2), it ensures that both compilations – for analysis and for bitcode generation – are identical (different compilers might apply different optimizations and have different default behavior). The resulting compilation command for the file `file.c`, generating `file.bc`, would look like this:

[2]Joern's old documentation (unavailable): https://docs.joern.io/llvm2cpg/getting-bitcode.

```
clang -emit-llvm -g -grecord-command-line -fno-inline-functions \
-fno-builtin {D2A_compiler_args} -c file.c
```

Thanks to this compilation process and the information from `compiler_args`, the definition and application of macros are successfully achieved. This addresses the previously unconsidered problem of conditional compilation, which was mentioned at the beginning of this chapter.

At this stage, the use of the inference pipeline, described in more detail in Section 4.2, might seem applicable. It is capable of generating LLVM bitcode for any C/C++ project. However, D2A consists of 6 projects, and within a single project, the samples are not made from the same version, but from thousands of different versions of the given project. The inference pipeline would thus need to be executed separately for each of these versions, which is computationally infeasible. The generated LLVM bitcode for each sample would be vast, and most of it would later be removed during program slicing. Instead, information from D2A and the git repositories of projects from D2A is used to compile only the necessary files on specific project versions.

For simplicity, consider the transformation of a single project within D2A. For each sample, it is necessary to restore the project repository to the version (commit) in which the error appears, which can be obtained from the D2A attribute `commit`. Then, the names of the files that need to be transformed into LLVM bitcode are extracted from D2A. These files are then compiled to generate LLVM bitcode. However, for this process to be fully automated, successful compilation of at least the required files across all required commits must be ensured. Proper configuration data, all dependencies, generated data (e.g., C headers), etc., are needed for successful compilation. All these elements change with software development, and automating LLVM bitcode generation requires manual adjustment to the specific project (more in Section 5.2).

Once a set of LLVM bitcode files is generated for each sample, these files need to be merged into a single one. This requirement stems from the requirements of the LLVM-Slicer tool (see Section 2.5). The tool `llvm-link` [56] is used for this purpose, which, despite its name, is not involved in the typical linking process of compilers. `llvm-link` merely combines multiple LLVM bitcode files into a single one while preserving the LLVM bitcode format. The tool `llvm-link` was chosen based on recommendations in the documentation of the LLVM-Slicer tool [12]. The output of the bitcode generation phase is, for each sample in the dataset, a single LLVM bitcode file containing the transformed source code of all files relevant to that sample. Additionally, the D2A dataset is filtered to include only supported error types.

4.1.2 Graph Construction

The input to the graph construction phase, as shown in Figure 4.3, consists of the filtered D2A dataset and an LLVM bitcode file for each sample. The output for each sample is a CPG, extended with additional information (the format of the output data is described in Section 4.1.3). The output graphs are stored in the CSV format for the Neo4j database. Additionally, a script in the Cypher language is included with each sample to load the respective graph into the database [88]. Although Neo4j is commonly used for storing and querying graph data, it is not used in this thesis.

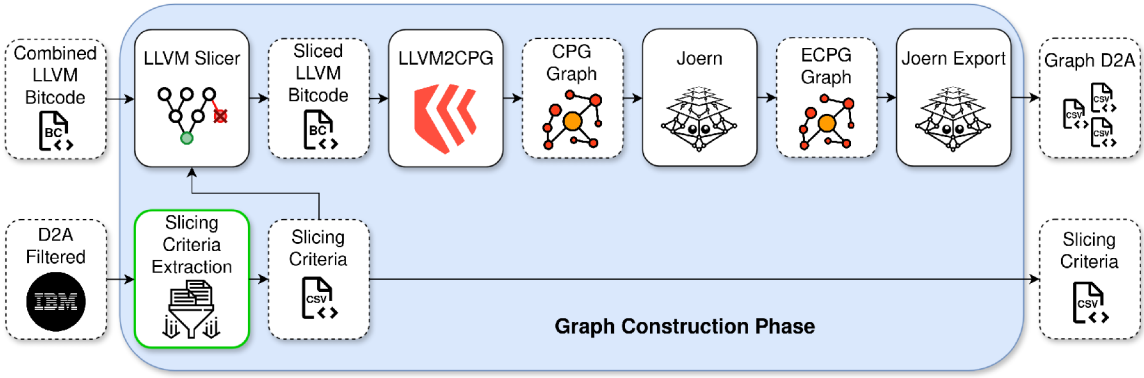

Figure 4.3: The figure shows a schematic of the graph construction phase, which creates a raw ECPG for each input D2A sample. Dashed lines represent the intermediate products and generated data. A green outline highlights the tools developed in this thesis. Icons were taken from [100, 99, 107, 101, 87].

Program Slicing

First, for each sample, it is necessary to extract information required for program slicing performed by the LLVM-Slicer (see Section 2.5) from the D2A dataset. This information includes:

- The **entry point function** – the function in which the program slicing should start (the top-most function in the bug trace).
- The **file** – the name of the file where the error is located.
- The **function** – the name of the function in which the error is located.
- The **line** – the line number on which the error is located.
- The **variable** (optional) – the variable related to the error (relevant only for certain types of errors).

These details form the so-called slicing criteria. The challenge with extracting slicing criteria is that each type of error has a different format, and these details cannot be uniformly obtained from all samples. The name of the entry point function is an exception and can be retrieved for all samples from the `procedure` attribute (both from D2A and Infer analysis output). The entry function is the highest-level function in the call graph[3] among all the functions mentioned in the Infer bug trace. Practically, this means that anything above this function is not important for the manifestation of the error. If the entry function itself is called with the correct parameters and in the right context, it must lead to the reported error (assuming it is a true positive). This is crucial information for program slicing, as it means that everything above the entry function can be discarded (which is exactly what LLVM-Slicer does), because it is not useful for the future classification of true positives/false positives.

[3] A **Call Graph** is an oriented graph in which nodes represent functions and edges represent calls between these functions. It can be considered a substructure of the control flow graph to some extent.

In total, 14 types of errors are supported, as shown in Tables C.1 and C.2. However, some error types are similar enough that slicing criteria can be extracted in the same way, resulting in the following groups:

1. `NULLPTR_DEREFERENCE` – this group contains only the identically named type.
2. `INTEGER_OVERFLOW` – includes all `INTEGER_OVERFLOW_X` where $X \in \{\texttt{L1}, \texttt{L2}, \texttt{L5}, \texttt{U5}\}$.
3. `INFERBO_ALLOC_MAY_BE_BIG` – contains only the identically named type.
4. `UNINITIALIZED_VALUE` – again contains only the identically named type.
5. `BUFFER_OVERRUN` – includes all `BUFFER_OVERRUN_X` where $X \in \{\texttt{L1}, \texttt{L2}, \texttt{L3}, \texttt{L4}, \texttt{L5}, \texttt{U5}\}$.
6. `NULL_DEREFERENCE` – the last group contains only the identically named type.

The `NULLPTR_DEREFERENCE` Group For errors of type `NULLPTR_DEREFERENCE`, three different formats of the Infer output were found in the D2A, distinguishable by the `qualifier` attribute [39], which contains a brief description of the error and shares the same name as in the JSON output of Infer:

1. `'call to ‘put_bits()‘ eventually accesses memory that is the null pointer on line 543 indirectly during the call to ‘init_put_bits()‘.'` – e.g., sample with `id: ffmpeg_8e48b53d696b53cef2814548e4d0693387e875ea_1`.
2. `'accessing memory that is the null pointer on line 3191 indirectly during the call to ‘av_malloc()‘.'` – e.g., sample with `id: ffmpeg_6a30264054cc320fe610c072c71d008f7e3c3efb_1`.
3. `'accessing memory that is the null pointer on line 315.'` – e.g., sample with `id: ffmpeg_9c908a4c99e0498dd26bd1de84ff085ac8e73e4a_1`.

For Cases 2) and 3), the `file`, `function`, and `line` information in the `bug_info` attribute are correct [39]. However, for Case 1), the same information only contains the location of the function call where the dereference occurs – in this case, `put_bits()`. The correct error location needs to be obtained from the last step of the `trace` attribute (bug trace), as shown in Listing 4.1 which shows this last step for Case 1). For Cases 2) and 3), the locations in the `bug_info` and the last `trace` step coincide, thus uniformly extracting information from the last bug trace step is feasible for all three formats. Additionally, all three formats share the same last step with `description` – `invalid access occurs here`. The only difference is that in `trace`, `function` is named as `func_name` and `line` must be extracted from `loc`, as seen in Listing 4.1.

For `NULLPTR_DEREFERENCE` errors, it is indeed sensible to consider extracting the variable name because null dereferences typically occur on a variable. However, they can theoretically occur on a constant like `(NULL)` or a general expression, such as `(p-p)`. When the null dereference occurs on a variable, it would be best to extract the name of this variable along with the error location. A more precise slicing criterion would allow for more accurate program slicing, thereby removing more unnecessary information from future graphs. Unfortunately, the variable name does not appear in any of the described

```
"trace": [
    // ... other trace steps ...
    {
        "idx": 16,
        "level": 2,
        "description": "invalid access occurs here",
        "func_removed": null,
        "file_removed": null,
        "file": "libavcodec/put_bits.h",
        "loc": "179:9",
        "func_name": "put_bits",
        "func_key": "libavcodec/put_bits.h@139:1-189:2",
        "is_func_definition": true,
        "url": "https://github.com/FFmpeg/FFmpeg/blob/5962f6b0da037da30
            fcc848331afa6a081a4eb09/libavcodec/put_bits.h/#L179"
    }
   ]
```

Listing 4.1: The last step of the `trace` [39] for a `NULLPTR_DEREFERENCE` error taken from the sample with `id: ffmpeg_8e48b53d696b53cef2814548e4d0693387e875ea_1` on the FFmpeg project. The listing demonstrates the format of storing a bug trace in D2A.

formats of `NULLPTR_DEREFERENCE` in the Infer output (and thus not in D2A either). The only clue available is the column, which can be extracted from the `loc` attribute (see Listing 4.1). Knowing the line and column where the dereference occurs might seem to simplify the extraction of the variable name. However, it is necessary to distinguish when the dereference is on a variable, a constant, an expression, or a macro. Distinguishing a variable can be done straightforwardly using the C language naming rules for variables – a name can only follow the pattern `[a-zA-Z_][a-zA-Z0-9_]*`, and anything else cannot be a variable and thus should not be extracted. However, macros can be named the same as variables (and typically are). Therefore, distinguishing between a variable and a macro is non-trivial and would require at least preprocessing by a compiler and subsequent adjustment of the error position, as macro expansion can change both the line and column. For now, the extraction of the variable name for `NULLPTR_DEREFERENCE` will be left for future improvements.

The `INTEGER_OVERFLOW` Group For the `INTEGER_OVERFLOW` error types, two formats were identified, which can again be distinguished using the `qualifier` attribute:

1. `([0, 8] - [0, 8]):unsigned32.`
   – e.g., sample with `id: ffmpeg_1542087b54ddf682fb6177f999c6f9f79bd5613f_1`.
2. `([0, 1] - 1):unsigned32 by call to ‘avfilter_unref_buffer‘.`
   – e.g., sample with `id: ffmpeg_ca5973f0bfac4560342605f8a52efc88b4f4dbd3_1`.

For Case 1), location information can be obtained directly from `bug_info`. For Case 2), a similar problem arises as with `NULLPTR_DEREFERENCE` Case 1) – `bug_info` contains only the location of the function call, in this case, `avfilter_unref_buffer`, where the integer

overflow/underflow occurs. It is necessary to extract information from the `trace` attribute. To cause an overflow/underflow, two operands are needed (with the exception of operators like `++`, which in terms of value change is equivalent to `+1`), and each operand can either be a variable or an expression. Although LLVM-Slicer is capable of slicing based on multiple criteria (in this case, both operands), to ensure the resulting graph is complete, it is necessary to include the operation itself. Thus, slicing must be based on the entire line, as slicing by expression is not yet supported (to the best of the author's knowledge).

The `INFERBO_ALLOC_MAY_BE_BIG` Group The `INFERBO_ALLOC_MAY_BE_BIG` group contains only a single error type of the same name, and only one format was identified:

1. `Length: [0, 2147483631] by call to ‘av_dup_packet‘.`
   – e.g., sample with `id: ffmpeg_c36d9fb10c31c6835d01232fddff6932a3ce347f_1`.

Similar to `NULLPTR_DEREFERENCE` Case 1), it is necessary to extract the location from the last step of the `trace` because the `bug_info` points to a function call in which the error occurs, in this case, `av_dup_packet`. The correct location points to a function call that allocates memory, such as `malloc`, `realloc`, etc. If the call appears as `realloc(ptr, size)`, ideally, it is preferable to slice directly by `size` because its value is of interest. However, the call can often appear as `realloc(ptr, str_len + 1)`, and in this case, the value of the entire expression is needed. Of course, it is possible to extract the name of the variable only in certain cases, but there are still the previously mentioned issues with macros and also with detecting the correct argument. From the output of Infer, it is not possible to determine which argument it concerns. With functions like `malloc` or `realloc`, the argument is known from their definitions, but it is necessary to consider cases where, during Infer analysis, custom models are created, and theoretically, any function could be considered an allocation function. For these reasons, for this type of error, slicing is only done by the line. Moreover, allocation functions typically are not very large, so including their code in the graphs does not represent too much unnecessary data.

The `UNINITIALIZED_VALUE` Group The `UNINITIALIZED_VALUE` group also contains only a single error type of the same name. Two formats have been identified, which can again be distinguished using the `qualifier` attribute:

1. `The value read from ret was never initialized.`
   – e.g., sample with `id: ffmpeg_ed80423e6bcfe18cca832b74dcc877427f8cf346_1`.

2. `The value read from pix[_] was never initialized.`
   – e.g., sample with `id: ffmpeg_1f62bae77d6ced3b79deaa8ce5ba3381fd4a541d_1`.

Neither format includes additional information in the `trace`, so the information is solely from `bug_info`. The location is correct for both formats. Case 1) concerns uninitialized variables, where it makes sense to slice also by the variable because the specific variable itself is of interest. Moreover, the variable's name can be easily obtained from the `qualifier`. Case 2) concerns uninitialized arrays (or items in an array), where the situation is more complex because it is not possible to obtain the access index into the array from the Infer output. It could be extracted from the code, but problems such as slicing by expressions

and macros arise. If slicing is done only by the line, then information about the index would be included in the output. However, experience from checking outputs from Infer (especially in [3]) suggests that if an array item is uninitialized, it is because the entire array was not initialized. For these reasons, even for Case 2), the variable name is extracted – in this case, it is always an array, and the slicing is done with respect to the array.

The `BUFFER_OVERRUN` Group For the `BUFFER_OVERRUN` error types, two formats have been identified, which can again be distinguished using the `qualifier` attribute:

1. `Offset: [0, 15] Size: 4.`
   – e.g., sample with `id:` `ffmpeg_61d490455ade68a02dfdcfdb172ba3ded2fe0f9d_1`.
2. `Offset: [1, 4] Size: 4 by call to ‘filter_mb_mbaff_edgecv‘.`
   – e.g., sample with `id:` `ffmpeg_0f5e5ecc888af015015f2ce1211a066350fbe377_1`.

For Case 1), the information in `bug_info` is correct. For Case 2), the location needs to be taken from the last step of the `trace` again. Neither format in the Infer output specifies the name of the array or the index name (if it involves a variable rather than an expression). For these types of errors, both the name of the array and the index are necessarily required. To obtain both names, a more complex extraction method from the source code would again be necessary. Hence, for this type of error, slicing is currently done by the line.

The `NULL_DEREFERENCE` Group The `NULL_DEREFERENCE` group contains only a single error type of the identical name. `NULL_DEREFERENCE` and `NULLPTR_DEREFERENCE` are semantically identical, with the difference lying in the Infer plugin that produced them – Bi-abduction (`NULL_DEREFERENCE`) and Pulse (`NULLPTR_DEREFERENCE`). Since they originate from different plugins (which may have issues with different language constructs leading to varying patterns of true and false positives) and also have different formats, it makes sense to list them separately. Two formats have been found, distinguishable by the `qualifier` attribute:

1. `pointer ‘filter‘ last assigned on line 3191 could be null and is dereferenced at line 3194, column 9.`
   – e.g., sample with `id:` `ffmpeg_15ae526d6763d8e21833feb78680ee3571080017_1`.
2. `pointer ‘null‘ is dereferenced by call to ‘ff_sdp_write_media()‘ at line 2538, column 5.`
   – e.g., sample with `id:` `ffmpeg_a94ada4250ff1d9e6101c910fe71dde6c3b5e485_1`.

Ideally, for this type of error, it would be desirable to slice by the variable name (if the incorrect dereference occurs on a variable). For Case 1), both the location information and the variable name can be obtained directly from `bug_info`, as the variable name is mentioned in the `qualifier`. For Case 2), `bug_info` contains only the location of the function call within which the incorrect dereference occurs. If the `qualifier` in Case 2) contains a variable name (instead of `null`), it is not the variable on which the dereference occurs but a variable whose value is passed to the called function. Unfortunately, in the `trace`, it is not possible to determine which step represents the incorrect dereference because

most steps lack a `description`. In some cases, the last step in `trace` represents the incorrect dereference, but in some others, it does not. Unfortunately, it is also not possible to distinguish between these types. For these reasons, for Case 2), slicing is done only by the line of the called function. This ensures that the error is included in the graph, even if it is deeper in the call graph. However, this introduces a significant ammount of unnecessary information.

The `adjusted_bug_loc` Attribute An important note is that any sample, regardless of the type of error, may contain the `adjusted_bug_loc` attribute in D2A. The attribute adjusts the location of the error if the `bug_info` – extracted directly from Infer's output – does not precisely pinpoint the error's location. The `adjusted_bug_loc` was derived using the same principles previously described for each group of error types. This raises the question of why not directly use `adjusted_bug_loc`. In the training pipeline, it is indeed possible to use these data, but in the inference pipeline, this information is no longer available because it only operates with Infer's output on real-world software, not with D2A. Therefore, a similar method of extracting precise error locations from the Infer report will eventually need to be designed and implemented for real-world applications. Utilizing it also in the training pipeline has additionally allowed verification of whether the author of this thesis and the authors of D2A agree on the method for extracting the exact location of errors – this has been verified across all samples of supported error types.

The Application of Program Slicing The extracted slicing criteria, along with LLVM bitcode, form the input for LLVM-Slicer (see Section 2.5). The output is a sliced LLVM bitcode according to the input slicing criteria. The purpose of program slicing is to remove parts of the graphs that do not influence the occurrence of the error. Consequently, this effectively reduces the size of the resulting graph and eliminates unnecessary information, which should facilitate and speed up the learning process for GNN models. LLVM-Slicer was chosen based on a recommendation by Ing. Viktor Malík. Upon verification, it was found to meet all the requirements, particularly in terms of input and output formats. An alternative, such as the tool `llvm-slicing`[4], is no longer maintained.

Program slicing at this stage of the pipeline also allows for specifying slicing criteria in relation to the original code, which is more appropriate than specifying them later in the CPG and slicing using tools like Joern [88]. Theoretically, it should be possible to identify slicing criteria in the output CPG graph thanks to debug information (see Section 4.1.1) attached to individual nodes, which can map certain LLVM constructions back to the original code. However, this information may be lost during CPG construction, as discussed in Section 2.6. Consequently, it becomes challenging to accurately map CPG nodes back to the original code, risking the incorrect construction of the node set intended for slicing criteria.

Generation of Extended Code Property Graphs from LLVM Bitcode

Sliced LLVM bitcode serves as the input to the LLVM2CPG tool (see Section 2.6), which generates CPGs. These CPGs are then processed by the Joern tool (see Section 2.7), whose task is to:

[4] **llvm-slicing**'s repository: https://github.com/zhangyz/llvm-slicing.

1. convert CPGs from binary to the CSV format,
2. create Extended CPGs (ECPGs) by adding additional layers such as information about types, files, functions, and more (see Section 4.1.3).

Joern is utilized solely as a CLI[5] tool in this thesis. It takes a binary CPG and a script with a list of commands to execute, primarily load and save operations, as Joern automatically constructs additional layers upon loading. The resulting ECPGs are saved again in a binary format. The final step involves converting the binary format into the easily usable CSV format using a Joern sub-tool – `joern-export` [88], which creates a directory with CSV files containing:

- CSV header file,
- CSV data file (without header),
- Cypher script for importing into the Neo4j database.

The CSV header file is kept for each sample, even though it might seem unnecessary. The reason lies in Joern's non-deterministic behavior regarding the columns generated on different machines. For instance, the header for `METHOD` nodes (see Section 4.1.3) in httpd (true positives) and openssl (true positives) – openssl has two additional columns, which, however, contain no useful information.

The Joern tools (and Joern Export) were chosen due to their frequent use in the field of GNNs [9, 33, 75, 91, 95]. LLVM2CPG is specifically recommended on the LLVM Project website for generating CPGs in combination with Joern [16].

4.1.3 Graph D2A

The D2A dataset, where each sample is transformed into an ECPG generated using the Joern tool, will henceforth be referred to as **Graph D2A**. Graph D2A is one of the main contributions of this thesis and, in combination with the original D2A, enables other researchers to create their own graph representations (based on CPGs) for GNNs and apply their own feature engineering. As mentioned earlier, ECPGs contain additional information compared to CPGs described in Section 2.4. Additionally, the CSV format facilitates further preprocessing of the dataset before inputting into GNNs. In this thesis, individual samples of Graph D2A will be referred to as raw ECPGs, precisely because they are in the CSV format and because the node/edge attributes are not yet processed or modified – they often lack, have an inappropriate format, and the format is not uniform (`int`, `float`, `string`, etc.). Processing raw ECPGs into a format suitable for training GNNs will be addressed in Section 4.1.4.

The complete output format of raw ECPGs is described in the automatically generated documentation of the Joern tool [93] (version 1.1), from which the following information is also taken. Like CPGs, raw ECPGs are directed, node-labeled, edge-labeled, multigraphs. The set of nodes that share the same label will be referred to as a *node set*, for future compatibility. All nodes within a single node set have the same set of attributes (although

[5] **Command Line Interface (CLI)**.

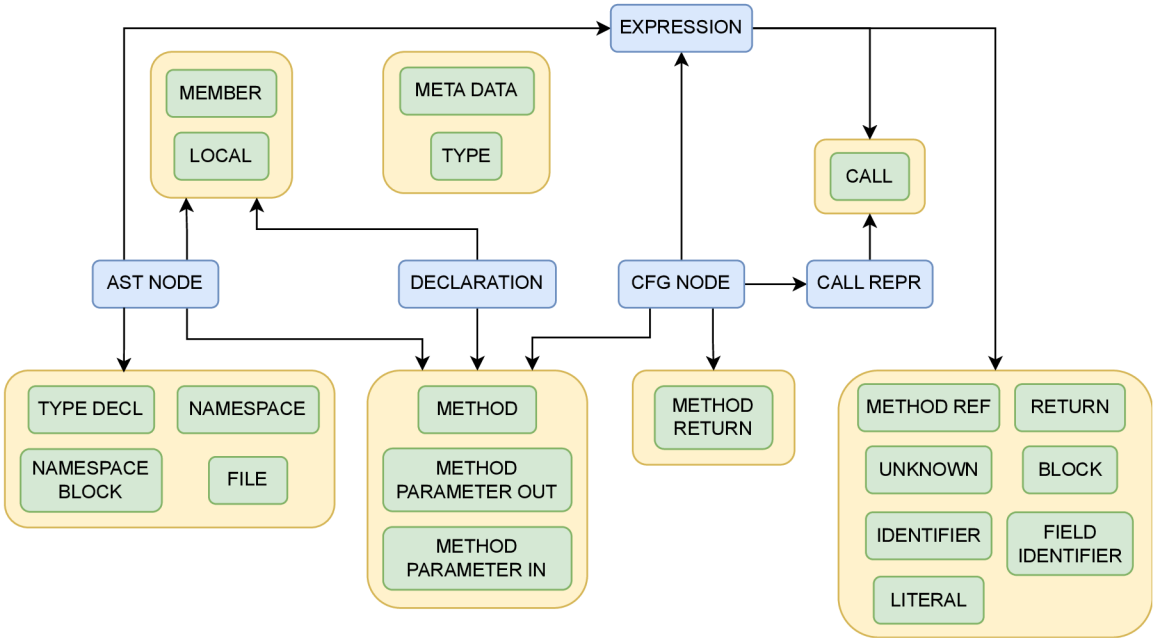

Figure 4.4: The figure shows the hierarchy of node sets in green (generated for LLVM IR) and the base class node sets in blue.

some values may be missing). Similarly, edges that share the same label will be referred to as an *edge set*. No edge sets (except `REACHING_DEF`) in raw ECPGs have attributes, but as mentioned in the following sections, it may make sense to move certain attributes from nodes to edges. Raw ECPGs consist of *layers*, where each layer can add additional node/edge sets or their attributes. Individual layers may be language-dependent, as Joern can generate ECPG graphs from any language for which a frontend is written. Currently, this includes languages such as **LLVM IR**, C/C++/C#, Java, JavaScript, Python, Kotlin, PHP, Go, Ruby, Swift, and more [93, 88]. The Joern layers that are **completely** missing for LLVM IR are the **Comment Layer**, **Finding Layer**, **TagsAndLocation Layer**, **Configuration Layer**, **Binding Layer**, and **Annotation Layer**. These missing layers will not be further described.

Since this thesis utilizes only LLVM IR (specifically LLVM bitcode) as Joern input, the following explanation includes only layers generated for LLVM IR. Although the Joern documentation is highly detailed, it does not describe all the attributes of the node sets. A complete list of these attributes is provided in Table C.4. The description of each attribute is discussed in detail in Section 4.1.4, where the removal of irrelevant attributes is addressed. Node sets are hierarchically organized – if node set `X` inherits from node set `Y`, it implies that `X` contains the same attributes as `Y` and adds some unique ones (typically). A visualization of the hierarchy of node sets (which are generated for LLVM bitcode) and base class node sets can be seen in Figure 4.4.

The MetaData Layer contains only one node set – `META_DATA`. In each graph, there is precisely one such node with `ID: 1`, containing information about how the graph was generated – e.g., input language, version, etc.

The FileSystem Layer includes information about the files from which the graph was generated. It specifically adds the node set `FILE`, where each node represents an input

source file. This layer also introduces an edge set `SOURCE_FILE`, which connects nodes from other node sets to `FILE` nodes based on their source file.

The Namespace Layer introduces the `NAMESPACE` node set, which resembles `FILE` and describes the namespace as known from programming. This layer also introduces the `NAMESPACE_BLOCK` node set, which groups code under a common namespace, defined using specific statements like `namespace` in C++ or `package` in Java.

The Method Layer includes declarations of methods, functions, and procedures (collectively referred to as 'functions' hereafter). This layer also includes their inputs and outputs but **does not contain their code**. Included in this layer are node sets:

- `METHOD` – information about a specific function.
- `METHOD_PARAMETER_IN` – represents the input parameters of a specific `METHOD` node.
- `METHOD_PARAMETER_OUT` – represents the output parameters corresponding to the inputs of a specific `METHOD` node.
- `METHOD_RETURN` – represents the return parameter of a specific `METHOD` node.

The Type Layer contains information about type declarations, type relationships, type instantiation, type hierarchies, parameterized types, and aliases. This layer introduces the following node sets:

- `MEMBER` – member of a structured type.
- `TYPE` – instance of a type.
- `TYPE_ARGUMENT` – argument used during parameterized type instantiation (e.g., Java `Generics`, C++ `templates`).
- `TYPE_DECL` – type declaration.
- `TYPE_PARAMETER` – formal parameter of parameterizable types.

Additionally, the layer provides the following edge sets:

- `ALIAS_OF` – alias relationship between a type declaration and a type.
- `BINDS_TO` – links type arguments to type parameters during type instantiation.
- `INHERITS_FROM` – inheritance relationship between type declarations and types.

The Ast Layer is the core of ECPGs, providing ASTs for all input code. AST nodes are linked into trees via the `AST` edge set, and sibling positions in the tree are specified using the `ORDER` attribute. The layer offers the following node sets:

- `AST_NODE` – template providing basic attributes of AST nodes.
- `BLOCK` – compound statement grouping multiple statements.
- `CALL` – function call.

- `CALL_REPR` – template for the `CALL` node set.
- `CONTROL_STRUCTURE` – control structure statements and jumps.
- `EXPRESSION` – template for any code fragment that can be evaluated.
- `FIELD_IDENTIFIER` – identifier of an element in an array.
- `IDENTIFIER` – identifier of a variable.
- `JUMP_LABEL` – jump label.
- `JUMP_TARGET` – any code location marked as a jump target.
- `LITERAL` – constant.
- `LOCAL` – local variable.
- `METHOD_REF` – function reference when passed as a parameter.
- `MODIFIER` – language-specific modifiers like `static`,`private`, `public`, etc.
- `RETURN` – `return` statement.
- `TYPE_REF` – reference to a type/class.
- `UNKNOWN` – other code fragments not classifiable into any of the above node sets.

The CallGraph Layer describes the relationships between function calls. This layer provides only the following edge sets:

- `ARGUMENT` – links `CALL` nodes to their arguments and `RETURN` nodes to the expressions they return.
- `CALL` – links `CALL` nodes to `METHOD` nodes.
- `RECEIVER` – links `CALL` nodes to the objects on which the method was invoked.

The Cfg Layer provides CFGs for all functions. This layer provides the `CFG_NODE` node set, which is also an `AST_NODE`. Therefore, all `CFG_NODE` are `AST_NODE`, but not all `AST_NODE` are `CFG_NODE`. Additionally, the layer adds the `CFG` edge set, which links `CFG_NODE` nodes in the direction of control flow (without distinguishing between true and false paths).

The Dominators Layer provides *dominator* and *post-dominator trees* [2] for all functions. These trees are closely related to the CFG Layer, as they identify sets of inescapable nodes in CFGs. The layer provides the following edge sets:

- `DOMINATE` – an edge indicating that the source node *dominates* the destination node.
- `POST_DOMINATE` – an edge indicating that the source node *post-dominates* the destination node.

The Pdg Layer provides PDGs for all functions. As defined in Section 2.4, a PDG should provide data dependency and control dependency edges. The Pdg Layer provides the `CDG` edge set, which provides control dependency edges (without distinguishing between true and false paths), and the `REACHING_DEF` edge set, which indicates that a variable (source node) reaches a specific point (target node) unchanged – an extension of data dependency edges.

The Shortcuts Layer provides a more explicit representation of certain properties using the following edge sets:

- `CONTAINS` – links nodes to the function (`METHOD` node) that contains them.
- `EVAL_TYPE` – links a node to its data type (`TYPE` node).
- `PARAMETER_LINK` – connects `METHOD_PARAMETER_OUT` nodes to their corresponding `METHOD_PARAMETER_IN` nodes.

The Base Layer provides the `DECLARATION` node set, which is merely a template for all declarations. Additionally, it provides the `REF` edge set, which indicates that an `IDENTIFIER` (source node) belongs to a specific node (target node), e.g., an identifier belongs to a local variable (`LOCAL` node).

An important note is that neither `CFG` nor `CDG` edges contain any information, which differs from the definition in Section 2.4. This can cause issues, especially when correctly modeling program branching. However, the required information is present in the graph, and branching can be modeled, as discussed in Section 4.1.4.

4.1.4 Feature Engineering

Graph D2A provides raw ECPGs in the CSV format. However, these graphs cannot be directly used to train GNNs. They first need to be transformed into a format suitable for model training, which is ensured by the feature engineering phase in Figure 4.5. The graph format is determined by the library chosen for implementing GNNs. In this thesis, TFGNN (TensorFlow GNN) is used, which is an open-source[6] extension of TensorFlow – one of the most widely used machine learning libraries. A relatively simple and commonly used dataset format within TF (not only for graph data) is TFRecord, which is designed to store sequences of binary data [84] – graphs, in this case. The input to the feature engineering phase is thus Graph D2A, and the output is a dataset in the TFRecord format. This transformation also includes feature engineering, which consists of the following steps:

1. **Feature Selection** – removing edge/node sets and their attributes that are not important for ranking false positives.
2. **Graph Optimization** – reducing the graph size while preserving crucial information.
3. **Node/Attribute Transformation** – some nodes/attributes need to be converted to another format or decomposed into multiple components.
4. **Feature Normalization** – it is beneficial to normalize features for more stable training.

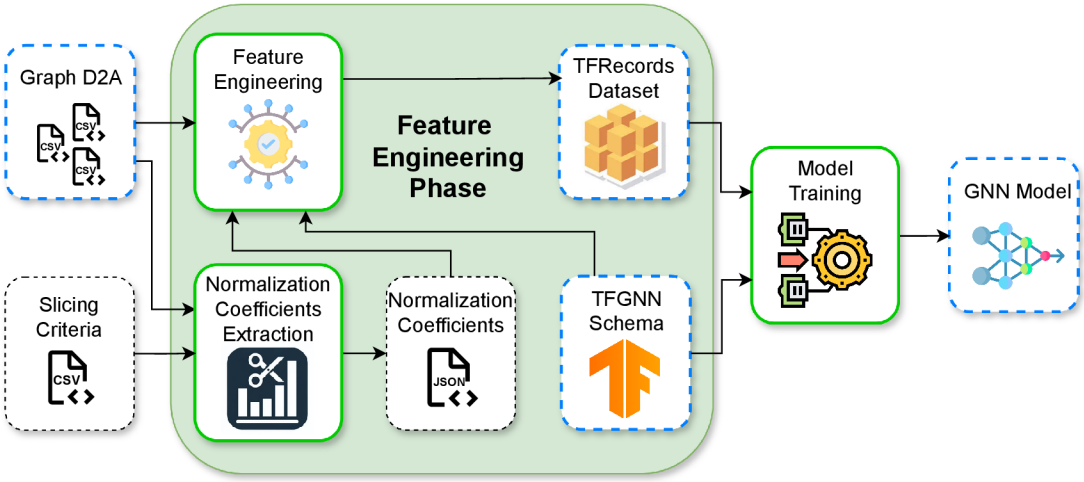

Figure 4.5: The figure shows a schematic of the feature engineering phase – the final phase of the training pipeline. Dashed boxes represent the intermediate products and generated data. A blue outline highlights the important outputs of the pipeline, and a green outline indicates the tools developed in this thesis. Icons were taken from [105, 66, 97, 104].

In feature engineering, the aim is to refine raw ECPGs to a state where models can be as small as possible while learning and generalizing effectively. The ultimate goal in reducing false positives is to train models that work for *cross-analysis* – training on one project while performing *inference* on a different, unseen project. None of the models compared in this thesis, specifically those from [94, 68], function within cross-analysis. Thus, in feature engineering, adjustments will be made that should help achieve cross-analysis for GNNs. The aim is to remove information from raw ECPGs that could lead the GNN model to *overfit* to individual samples or entire projects. These adjustments include, for instance, removing specific variable names or suppressing the original programming style (already managed by converting to LLVM IR). These modifications are described in detail later in this section.

Feature Selection

Feature selection is one of the most crucial parts of the training pipeline because it allows for the greatest reduction in graph sizes. Simultaneously, incorrect feature selection prevents the GNN models from efficiently extracting distinguishing patterns. This significant importance was the primary motivation for Graph D2A (raw EPCGs) to also be one of the outputs of this thesis. If feature selection was poorly executed, the resulting dataset would be unusable, providing only incomplete data for the task. Thanks to Graph D2A, experimenting with different feature selections in the future is possible.

It is important to choose node/edge sets and their attributes that best and most accurately describe the code from the perspective of potential errors. These optimized graphs should contain only the information needed to distinguish between true positives and false positives. Additionally, the aim is for models to generalize well to unseen projects; for instance, the model should not remember variable names and thus recognize specific pro-

[6] **TFGNN**'s repository: https://github.com/tensorflow/gnn.

jects/samples based on them. This section describes which node/edge sets and their attributes were chosen and why (also why some were not chosen). The following description includes only edge/node sets that were generated for LLVM IR (at least for a single sample). Node sets that did not occur (even if some additional information from their layer did, see Section 4.1.3) are `TYPE_ARGUMENT`, `TYPE_PARAMETER`, `CONTROL_STRUCTURE`, `JUMP_LABEL`, `JUMP_TARGET`, `MODIFIER`, and `TYPE_REF`. Template node sets (see Figure 4.4) are also not generated. The edges `RECEIVER`, `CONDITION`, `INHERITS_FROM`, and `BINDS_TO` are also not present in Graph D2A. Attributes not mentioned for a given node set, and that the node set does have (see Table C.4), contain only a single (or, for instance, two but useless) value across the entire dataset and thus do not provide any useful information. The following descriptions are based on the Joern documentation [93] and the examination of Graph D2A samples.

The `META_DATA` Node Set The node set `META_DATA` will not be used at all because all its attributes contain only a single value for all samples. The only exception might be `LANGUAGE`, which holds information about the language from which the ECPG was generated. This is useful when the system has multiple input languages, which theoretically could be true for the proposed system (thanks to converting the input language to LLVM IR). However, since each input language would first be compiled to LLVM IR, `LANGUAGE` would always have the value `'LLVM'`.

The `FILE` Node Set The node set `FILE` also contains only attributes with the same values across samples. The reason it does not contain information about files is LLVM Link – the input file to Joern with LLVM bitcode is always just one, so the node set `FILE` contains only a single node. Thus, the node set will be completely removed. Simultaneously, the edge set `SOURCE_FILE` will also be removed because it depends on the `FILE` node set.

The `NAMESPACE` Node Set The node set `NAMESPACE` will also be completely removed. LLVM IR does not have namespaces like other high-level languages. At best, the concept of namespaces can be discussed in relation to individual files. However, the same problem arises as with the `FILE` node set – all code is in a single file and thus in a single namespace. The attribute `NAME` contains only the values `<global>` and `llvm-link_global`. Analogously for the node set `NAMESPACE_BLOCK`.

The `METHOD` Node Set The node set `METHOD` will be used, specifically its attributes:

- `IS_EXTERNAL` – has values `true` and `false`, indicating whether the function's code is available (and therefore a ECPG) or not (dynamic library call).
- `ORDER` – the value is always `0`, but it is retained for later node set merging (see below in this section).
- `FULL_NAME` – the full name of the function (e.g., `malloc`). Ideally, this information should be removed to prevent the model from remembering functions from individual D2A projects, reducing the likelihood of successful cross-analysis. However, it is necessary to distinguish functions for which no code is available. Functions from

standard libraries are crucial to remember because, for instance, `malloc` or `free` are essential for detecting memory leak errors. Using `FULL_NAME` only for `IS_EXTERNAL` functions is logical because they often belong to standard libraries, avoiding project-specific function names. However, project-specific functions can also be dynamically linked, and standard functions can be linked statically. For now, `FULL_NAME` will be used for all functions, and the usage limited to some functions only is left for future work.

The following attributes contain some information but will not be used:

- `AST_PARENT_FULL_NAME` – for `METHOD` nodes, this is the name of the `NAMESPACE_BLOCK`, which does not contain any useful information, as mentioned before.
- `FILENAME` – analogously to `AST_PARENT_FULL_NAME`.
- `LINE_NUMBER` – information about which line the definition is on (may be empty), this information is not crucial for error detection.
- `NAME` – for `METHOD` nodes, it contains, like `FULL_NAME`, the function name.
- `SIGNATURE` – contains the function signature, which is potentially useful information. However, since the data types of individual nodes will be included later (including the arguments and return values of all functions) as separate nodes, the signature will be implicitly present in the graph structure, and the `SIGNATURE` attribute would be redundant.

The `METHOD_PARAMETER_IN` Node Set For the node set `METHOD_PARAMETER_IN`, only the `ORDER` attribute will be used (the order among siblings in the AST), which indicates the index/order of the parameter within the function declaration. The other attributes will not be used, namely:

- `CODE` – contains the parameter name, which is better removed to improve generalization between projects.
- `INDEX` – always has the same value as `ORDER` and expresses the same information, so it is unnecessary redundancy.
- `NAME` – contains the same value as `CODE` – will be removed.
- `TYPE_FULL_NAME` – information about the data type, but since data types will be modeled as separate nodes, this information is redundant.
- `IS_VARIADIC` – information on whether the parameter is variadic (denoted as `'...'` in C, e.g., in the `printf` function). This information will be discarded for future node set merging purposes, but it is still useful information.

The `METHOD_PARAMETER_OUT` Node Set Node set `METHOD_PARAMETER_OUT` will be completely removed because, for statically typed languages (like LLVM IR), it contains the same information as `METHOD_PARAMETER_IN` and would unnecessarily add redundant data. The edge set `PARAMETER_LINK` is also removed because it connects `METHOD_PARAMETER_IN` and `METHOD_PARAMETER_OUT`.

The METHOD_RETURN Node Set In the METHOD_RETURN node set, only the ORDER attribute will be used, mainly due to the later merging of node sets. The attributes CODE, DYNAMIC_TYPE_HINT_FULL_NAME (which can be empty), and TYPE_FULL_NAME typically contain the same information about the data type, which will be discarded for the aforementioned reasons.

The MEMBER Node Set For the MEMBER node set, only the ORDER attribute will be used, indicating the order within the defined structure. Other attributes will not be used:

- CODE – contains the name of the component. The name of the component is not important for distinguishing true positives and false positives; only its type and order matter.
- NAME – contains the same information as CODE.
- TYPE_FULL_NAME – types will later be expressed via nodes.

The TYPE Node Set The TYPE node set contains the attributes FULL_NAME, NAME, and TYPE_DECL_FULL_NAME, which contain the same information – the full name of the data type. Therefore, only FULL_NAME will be retained (although any of them could be used). Based on the name, the data type can be distinguished into multiple categories, such as integer, float, pointer, function signature, etc. More information can be found below in this section. Modeling data types using external nodes greatly simplifies other nodes that (where appropriate) carry their own type information in their attributes, which can then be removed. Overall, graphs will be smaller (in terms of data quantity in attributes) and simpler.

The TYPE_DECL Node Set The TYPE_DECL node set does not add any new information for LLVM IR compared to the TYPE node set. The node set could be useful for languages with parameterizable types or classes like C++ or Java. Therefore, this node set is completely removed. However, it must be removed carefully because it connects important parts of the graphs – TYPE nodes of structured types with their MEMBER nodes. More information on this is provided later in this section.

The BLOCK Node Set The BLOCK node set does not contain any useful information in its attributes. Its usefulness lies in how it connects other nodes in the AST. For instance, each function has its own BLOCK node that contains all top-level statements. BLOCK nodes are useful, for example, for determining variable scope and also as *latent nodes*[7] for passing information within GNN [76]. Again, for future node set merging, the ORDER attribute is retained, as well as the ARGUMENT_INDEX attribute. However, neither carries any useful information.

[7] **Latent node** – a node in the graph that does not contain any information itself and serves purely as a connection between other nodes.

The CALL Node Set In the CALL node set, only the attributes ORDER and ARGUMENT_INDEX will be retained. If the parent of the CALL node is another CALL node, then ARGUMENT_INDEX indicates the position among the function call arguments. If the parent is a BLOCK node, ARGUMENT_INDEX indicates the position among the commands contained in that BLOCK node. However, this information is not particularly important and will be removed later, but detection of this case can only be done by examining the graph, as described later in this section.

The FIELD_IDENTIFIER Node Set LLVM IR supports the array data type [55], so the FIELD_IDENTIFIER node set must be included. Again, the ORDER and ARGUMENT_INDEX attributes are retained due to the later merging of node sets, although they contain the same value across the dataset. Both values are always 2, because access to an array is modeled in ECPGs as a call to the getElementPtr operator, where the FIELD_IDENTIFIER is always the second argument. The CANONICAL_NAME attribute, which contains the name of the field, will not be included for reasons similar to those for the FULL_NAME attribute in the METHOD node set – the model should not remember samples/projects based on specific names.

The IDENTIFIER Node Set In the IDENTIFIER node set, only ARGUMENT_INDEX and ORDER are retained, which now contain valid values. Other attributes are not included:

- CODE – contains the name of the variable, which is not used for the same reasons as CANONICAL_NAME in FIELD_IDENTIFIER.
- COLUMN_NUMBER – potentially useful information, especially for refining pooling in the GNN head (see Section 4.1.5), but this is left for future improvements.
- LINE_NUMBER – same reason as COLUMN_NUMBER.
- NAME – contains the same information as CODE.
- TYPE_FULL_NAME – types are handled using the TYPE node set.

This may raise the question of how to identify IDENTIFIER nodes that refer to the same variable in the graph when their names are discarded. The answer lies in the LOCAL nodes and REF edges that connect nodes representing the same variable. This information is thus represented by the graph structure and not by the node attributes.

The LITERAL Node Set In the LITERAL node set, the ORDER and ARGUMENT_INDEX attributes are retained. Additionally, the CODE attribute, which contains the literal value, is also retained. This value can be an integer, floating point, string, array, structure, or any supported data type in LLVM IR [55]. Storing this value will require the creation of a special node set capable of accommodating these different formats (see later in this section). The COLUMN_NUMBER, LINE_NUMBER, and TYPE_FULL_NAME attributes are not used for the reasons mentioned earlier.

```
1 store i32 %5, i32* @x, align 4, !dbg !40 // write x
2 %10 = load i32, i32* @x, align 4, !dbg !27 // load x
3 store i32 %6, i32* @y, align 4, !dbg !42 // write y
4 %7 = load i32, i32* @y, align 4, !dbg !24 // load y
```

Listing 4.2: An example of a code in LLVM IR that demonstrates reading and writing to global variables.

The `LOCAL` Node Set In the `LOCAL` node set, only the `ORDER` attribute is used. Other attributes, such as `CODE`, `NAME`, and `TYPE_FULL_NAME`, are not used for the previously mentioned reasons. The usefulness of this node set lies mainly in connecting the `IDENTIFIER` nodes that represent the same variable. The problem is that `LOCAL` nodes only exist for local variables, not for global ones. LLVM IR can have global variables, but unfortunately, LLVM2CPG and Joern cannot properly encode them in the graph. Consider the read and write operations in Listing 4.2. It is clear which variable is being read/written to – the `x` and `y` variables. However, if ECPG is generated using LLVM2CPG and Joern, each access to a global variable is preceded by obtaining its address, and then the data is written/read to/from that address – the variable identifier is not used. Obtaining the address is modeled as a call (a `CALL` node) to the `addressOf` operator, which has a single operand that should contain the global variable identifier in this case. But as shown in Listing 4.3, the operand is of type `LITERAL` with value `0` and type `i32`. Therefore, global variables cannot be distinguished from each other. One possible way would be to use the debug info `!dbg !40` in the original LLVM IR (see Listing 4.2), which points to the exact location (line and column) in the original C code, and thus the name of the global variable could be extracted from there. However, this encounters the previously mentioned problem – variables versus macros. Extracting global variables is thus left for future improvements.

The `METHOD_REF` Node Set In the `METHOD_REF` node set, only `ARGUMENT_INDEX` and `ORDER` will be retained. The `COLUMN_NUMBER` and `LINE_NUMBER` attributes will not be retained for the reasons mentioned earlier. The `CODE` and `METHOD_FULL_NAME` attributes both contain the name of the method that the node represents. However, since `METHOD_REF` is connected to the `METHOD` node via `REF` edges, these attributes can be discarded.

The `RETURN` Node Set In the `RETURN` node set, `ORDER` and `ARGUMENT_INDEX` are used, which contain useful values. The location information `COLUMN_NUMBER` and `LINE_NUMBER` are discarded again.

The `UNKNOWN` Node Set In the `UNKNOWN` node set, only `ORDER` and `ARGUMENT_INDEX` will be retained, both containing valid values. The `CODE` attribute may contain clues about what the `UNKNOWN` node holds – typically the name of a data type or a signature. The `UNKNOWN` node itself does not provide any useful information, but it is typically deeply embedded in the graph and connects surrounding nodes. Because there are significantly fewer `UNKNOWN` nodes compared to other nodes, it will be retained.

```
// node set CALL
ID,LABEL,CODE,COLUMN_NUMBER,LINE_NUMBER,METHOD_FULL_NAME,TYPE_FULL_NAME
35,CALL,&0,16,10,<operator>.addressOf,i32*
46,CALL,&0,16,12,<operator>.addressOf,i32*
88,CALL,&0,5,7,<operator>.addressOf,i32*
96,CALL,&0,5,8,<operator>.addressOf,i32*

// edge set AST
START_ID,END_ID,TYPE
35,34,AST
46,45,AST
88,87,AST
96,95,AST

// node set LITERAL
ID,LABEL,CODE,COLUMN_NUMBER,LINE_NUMBER,TYPE_FULL_NAME
34,LITERAL,0,16,10,i32
45,LITERAL,0,16,12,i32
87,LITERAL,0,5,7,i32
95,LITERAL,0,5,8,i32
```

Listing 4.3: The simplified ECPG in the CSV format for the LLVM IR code from Listing 4.2, demonstrating reading and writing to global variables.

Edge Sets The edge set `ALIAS_OF` is discarded because aliases are resolved and the original type names are used during compilation and generating LLVM bitcode. Information about aliases is present in the graph through debugging information (see Section 4.1.1) in the form of `TYPE` nodes. However, these nodes are later removed as part of graph optimizations (see below in this section).

The edge sets `AST`, `CFG`, and `CDG` are, of course, retained because they form the core of the CPG.

The edge set `ARGUMENT` is retained because it connects `CALL` nodes to their arguments.

The `CALL` edge set connects `CALL` nodes to their corresponding `METHOD` nodes, thereby adding a call graph to the CPG. The AST itself is created for each function, but the trees are not interconnected, preventing message propagation during GNN computation. The `CFG` edges, along with `CALL` edges, connect these individual ASTs at semantically appropriate places.

The edge sets `DOMINATE` and `POST_DOMINATE` form dominator and post-dominator trees [2], which provide useful information but essentially express certain simple properties of the CFG more explicitly. Additionally, there are too many of these edges, so they will not be used. However, it would be beneficial to experiment with them in future work.

The `REACHING_DEF` edge set will not be used because there are too many of these edges in each graph. However, this is another very useful edge set that is worth experimenting with in future work.

The CONTAINS edge set will not be used because it also represents a relatively large number of additional edges. Furthermore, the information about which method a node belongs to can easily be obtained from the AST.

The EVAL_TYPE edge set is, of course, used to connect nodes with their types.

The REF edge set is also retained to link identifiers to the local variable they identify. It also connects METHOD_REF nodes and METHOD nodes.

Mandatory Attributes All node sets also contain the ID and LABEL attributes, which are not mentioned in the documentation. There are more undocumented attributes, such as CLOSURE_BINDING_ID in the LOCAL node set. However, since none of them are used, they are not mentioned in the text. A complete list of attributes for node sets in Graph D2A can be found in Table C.4. The ID attribute identifies a node within each Graph D2A sample, and the LABEL attribute contains the name of the node set. Both of these attributes are used, although the ID is more for implementation reasons, and the original ID is not present in the output TFRecords files, which instead use IDs from the TFGNN library (see Section 5.6). Each edge set (except REACHING_DEF, which contains an additional attribute) contains the following three attributes:

- START_ID – the source node of the directed edge.
- END_ID – the target node of the directed edge.
- TYPE – the name of the edge set.

Control Structures in Raw ECPGs The generated ECPGs do not contain the node sets CONTROL_STRUCTURE, JUMP_LABEL, or JUMP_TARGET, even though the original C source code uses them. The reason lies partly in the conversion to LLVM IR and partly in generating the CPGs. During the compilation to LLVM IR, all control structures (if, for, while, etc.) are simplified to jumps. Consider a simple if statement in C in Listing 4.4. The same code in LLVM IR is shown in Listing 4.5, where label 6: represents the true branch and label 7: the false branch of the original code. The tools LLVM2CPG and Joern did not generate JUMP_LABEL or JUMP_TARGET even though they are present. However, program branching information (which is crucial) can still be extracted from the CPG, specifically using CFG and CDG edges.

Consider a partial ECPG (only CFG and CDG edges and without the ret statement) in Figure 4.6, representing the LLVM IR code from Listing 4.5. If a node branches CFG edges (node 30, representing the value assignment to %5), the program flow branches at that node. The possible CFG paths are branches in the original code. These branches are connected to the condition that determines the program flow branching via CDG edges (from node 30).

From the graph, one can distinguish the true branch (assigning 1 to %1) from the false branch (assigning 2 to %1) because the true branch has **lower** ORDER value for the last node (node 34) before the paths merge (node 40) than the node from the false branch (node 38). This is because the true branch is always first due to the compilation. If the LLVM IR is manually modified and the branches are rearranged, it will no longer be possible to distinguish the true and false branches from the graph. This fact further demonstrates

```
1  if (z)
2      return 1;
3  else
4      return 2;
```

Listing 4.4: A simple `if` statement in the C language.

```
1    %5 = icmp ne i32 %4, 0, !dbg !16
2    br i1 %5, label %6, label %7, !dbg !18
3
4  6:
5    store i32 1, i32* %1, align 4, !dbg !19
6    br label %8, !dbg !19
7
8  7:
9    store i32 2, i32* %1, align 4, !dbg !20
10   br label %8, !dbg !20
11
12 8:
13   %9 = load i32, i32* %1, align 4, !dbg !21
14   ret i32 %9, !dbg !21
```

Listing 4.5: The code from Listing 4.4, but converted to LLVM IR.

that utilizing node sets and edge sets constructing the CPG is necessary. It also shows the importance of `ORDER` attributes in AST nodes.

The node sets and their attributes present in Graph D2A are listed in Table C.4. Attributes of individual node sets that were considered useful and selected during feature selection are also marked in the table.

Graph Structure Optimization

Before using individual graphs as inputs to GNNs, it is necessary to remove as many unnecessary and redundant nodes, edges, and attributes from the graphs as possible. The previous text dealt with the removal of information at the level of entire node sets, edge sets, and attributes. However, even within a single node set, there are nodes that do not add any useful information and can be removed, effectively reducing the graph and easing the learning process of the GNNs. Furthermore, adjustments are required to ensure certain properties arising from the use of GNNs and the TFGNN library.

To propagate information correctly within the graph during GNN computation, the graph must consist of only one WCC (see Section 2.3). For GNNs where message passing follows the direction of the edges, it is also necessary to ensure the correct orientation of the edges. Using TFGNN requires creating a so-called TFGNN schema [77], which describes the graph structure: node sets, edge sets, their attributes, and attribute data types. In the TFGNN schema, all nodes in a node set must have the same attributes (similarly for edges) – this already applies to raw ECPGs. Furthermore, each edge set must have a fixed source node

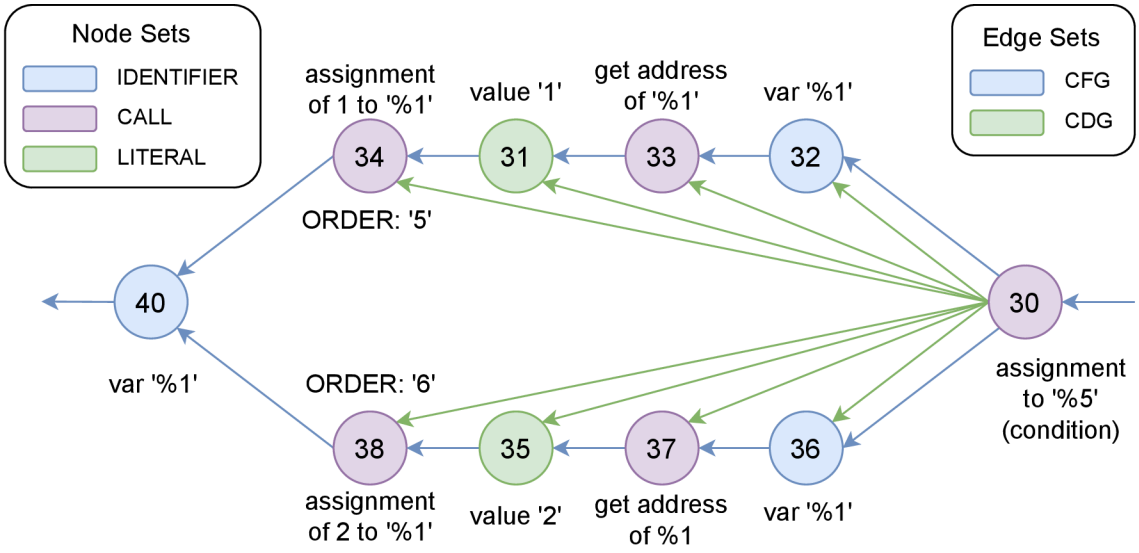

Figure 4.6: A partial (only `CFG` and `CDG` edges and without the `ret` statement) ECPG demonstrating branching in LLVM IR from Listing 4.5.

set and target node set – unfortunately, this is not the case with raw ECPGs. For example, `AST` edges connect `BLOCK`, `CALL`, `LOCAL`, `IDENTIFIER`, and other types of nodes.

The following transformations are described in this section:

1. removal of unnecessary and redundant information,
2. creating graphs with only one WCC,
3. ensuring fixed source and target node sets for all edge sets.

The removal of unnecessary and redundant information and ensuring a single WCC are closely related because removing some unnecessary nodes also removes unnecessary separate WCCs. This step consists of the following sub-steps, which must be performed in the given order:

1. removal of invalid nodes within the AST,
2. removal of WCCs (only in the AST) consisting only of `BLOCK` nodes,
3. removal of AST leaf `BLOCK` nodes,
4. filtering out unnecessary `ARGUMENT` edges,
5. removal of AST children of external functions,
6. removal of unused functions,
7. filtering out unnecessary `EVAL_TYPE` edges,
8. removal of the `TYPE_DECL` node set,
9. filtering out unused `TYPE` nodes.

Removal of Invalid Nodes The removal of invalid nodes occurs only within the AST – considering only `AST` edges and ignoring others. Invalid nodes are considered all those that were removed during feature selection and have an `AST` edge leading to or from them (e.g., a standalone `META_DATA` node is not part of the AST). If these nodes were simply removed (along with their edges), the AST they belong to would be split. Therefore, they need to be removed based on their position in the AST as follows:

1. Leaf – the invalid node, along with its edges, can be simply removed.
2. Root – the invalid node is replaced with a valid `BLOCK` node to ensure that the AST has exactly one valid root.
3. Inner node – the AST children of the invalid node are connected to the AST parent of the invalid node.

Since no node types inheriting from `CFG_NODE` are removed during feature selection, it is not necessary to connect `CFG` edges in the case of inner nodes because there should not be any, according to the documentation. Similarly, `CDG` edges do not make sense for any invalid nodes. An exception is the removed `METHOD_PARAMETER_OUT` node set, which inherits from `CFG_NODE` (see Figure 4.4), but no `CFG` or `PDG` edges leading to or from these nodes were found in Graph D2A. Therefore, only `AST` edges need to be reconnected.

After removing the invalid nodes, the graph is composed of one or more ASTs – one AST for each function.

Removal of `BLOCK` WCCs After removing the invalid nodes, it is necessary to remove WCCs entirely composed of `BLOCK` nodes. These are latent nodes, and WCCs entirely composed of latent nodes contain no useful information. Such WCCs can arise, for example, from ASTs entirely composed of invalid nodes, as the previous algorithm would convert the invalid root into a valid node and remove the other invalid nodes. This would result in an AST with only a `BLOCK` node. The case where a WCC consists only of `BLOCK` nodes is relatively rare.

Removal of Leaf `BLOCK` Nodes All leaf `BLOCK` nodes are also removed from all ASTs. The reason is that `BLOCK` nodes are used to cluster other nodes – if they have no AST children, they are unnecessary.

Filtering of `ARGUMENT` Edges Next, the `ARGUMENT` edge set is filtered to keep only edges that have a `CALL` node as their source. In other words, the `ARGUMENT` edges between `RETURN` nodes and the expressions they return (nodes inheriting from `EXPRESSION`) will be removed. This adjustment is made to move the `ARGUMENT_INDEX` into the `ARGUMENT` edges later. For `ARGUMENT` edges from `RETURN` nodes, it does not make sense to talk about the index of arguments (there is always only one for LLVM IR [55]), so these edges are removed.

Removal of `AST` Children of External Functions All AST children of `METHOD` nodes representing external functions (having the value `True` in the `IS_EXTERNAL` attribute) are then removed. The removed nodes for each function are one `METHOD_RETURN` and

N METHOD_PARAMETER_IN, where N is the number of function parameters. For external functions, this information is not useful because the information about input and output parameters is already present when calling the function (CALL node and its AST children). METHOD_PARAMETER_IN serves only as a link between arguments from the call site and the use of parameters within the function body for non-external functions. Similarly, METHOD_RETURN serves as an abstraction of all RETURN nodes (typically only a single `ret` statement in each function, see Listing 4.5) in the function body – it is connected to them.

Another reason for removing AST children of external functions is that for many operators, EVAL_TYPE edges led from the METHOD_PARAMETER_IN and METHOD_RETURN nodes to a TYPE node, whose type was ANY – which does not provide any additional information. By removing these AST children, these EVAL_TYPE edges are also removed, leading to the removal of the TYPE node with the ANY value. Thus, the data type ANY will not need to be considered.

Removal of Unused Functions If an unused function is found – in other words, if there is a METHOD node in the graph without incoming CALL edges – it is also removed. This can happen because each graph implicitly includes at least the function `llvm.dbg.declare`, which is part of the debug information [55]. There may also be some unused operators or other default global functions.

Filtering of EVAL_TYPE Edges The next step is the removal of EVAL_TYPE edges leading from the METHOD, BLOCK, and METHOD_REF nodes. All these node sets contain information about the data type, which do not need to be retained. For BLOCK nodes, this represents the return type of the entire block – in some languages (typically functional), this information is useful, but in C/LLVM IR, this value is irrelevant as it merely indicates the data type of the last statement in the given block. For METHOD and METHOD_REF, it represents the signature, which (as previously mentioned) is expressed through the data types of the function's inputs and outputs.

Removal of the TYPE_DECL Node Set In feature selection, it was mentioned that the TYPE_DECL node set needs to be removed in a specific way because it connects structures and their elements. If a TYPE node is a structure for which its MEMBER nodes are known (if only a pointer to it is used, the elements may not be known), then this TYPE node is connected by a REF edge to a TYPE_DECL node, from which AST edges lead to individual MEMBER nodes, as shown in Figure 4.7. Each TYPE_DECL is then removed such that if it has any AST MEMBER children, these MEMBER (target node) nodes are connected to the TYPE node (source node) using new CONSISTS_OF edges, and the TYPE_DECL is removed along with all edges leading to or from it. If the TYPE_DECL has no AST MEMBER children, it can simply be removed.

Filtering of Unused TYPE Nodes Unused TYPE nodes are then removed. An unused TYPE node has no incoming EVAL_TYPE edges. This removal process is iterative – it iterates until there are no TYPE nodes without incoming edges. This ensures that nested or recursive structures are also removed. One might ask what happens if a TYPE node has a self-loop (loop) – a recursive structure (or two or more mutually recursive structures). The answer lies in the representation of such structures – a recursive structure cannot contain itself, only

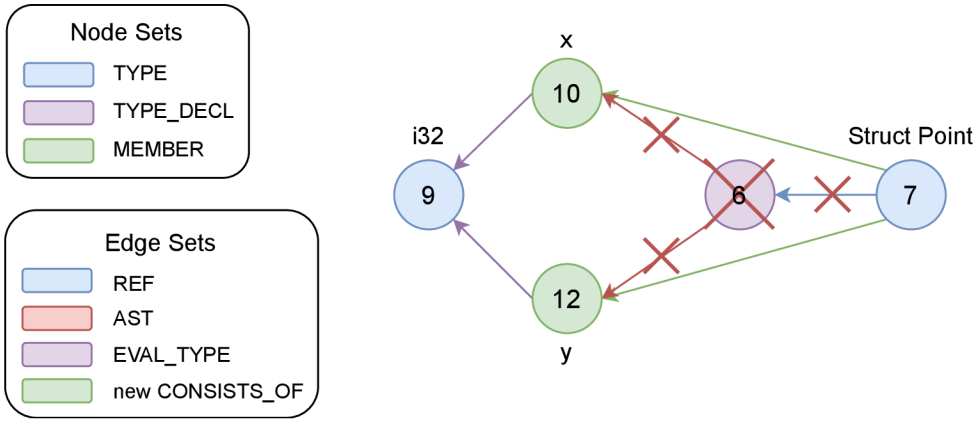

Figure 4.7: The figure shows the connection of `TYPE_DECL` nodes in ECPG when modeling structured data types.

a pointer to itself (similarly for mutual recursion). Self-loops and loops are not possible for `TYPE` nodes because `MyStruct` and `MyStruct*` are two different types, represented by two `TYPE` nodes. For this reason, the information about the individual elements of `MyStruct` does not need to be present in the graph if only its pointer `MyStruct*` is used and its elements are not accessed – information about the elements would be redundant.

As a result of these modifications, the graph is composed of a single WCC. When using bi-directional GNNs, information can be propagated between all nodes. For directional GNNs , it is still necessary to correctly orient the edge sets, as described below.

The Edge Set Condition The condition set by the TFGNN schema that each edge set must have a fixed source node set and target node set is currently not met. Examples include the basic `AST` edges that start and end in different node sets. This condition can be met in two basic ways:

1. Splitting all edge sets that do not meet the conditions into smaller edge sets to meet the condition. The problem with this solution is that the number of possible sub edge sets is up to $|N|^2$ where N is the set of node sets that can appear on either side of any edge in the given edge set – because it is necessary to cover each combination of node sets. Of course, some combinations are not possible, such as a `METHOD` node not having direct AST children of the type `CALL`, so practically, there are fewer combinations. Adding the fact that even `CFG` and `PDG` also connect a large number of node sets, like `AST`, this number increases significantly. This results in dealing with tens to hundreds of sub edge sets. The fact that it would be necessary to define them manually in the TFGNN schema, and the fact that the more edge sets there are, the more complex the GNN model, shows that this number of sub edge sets is unsustainable. The principle is demonstrated in Figure 4.8.

2. Some existing node sets can be merged so that all edge sets meet the required condition. This method simplifies the TFGNN schema (there will be fewer node sets) but requires that all node sets that will be merged into a super node set have the same attributes. For this reason, potentially useful attributes were discarded and some un-

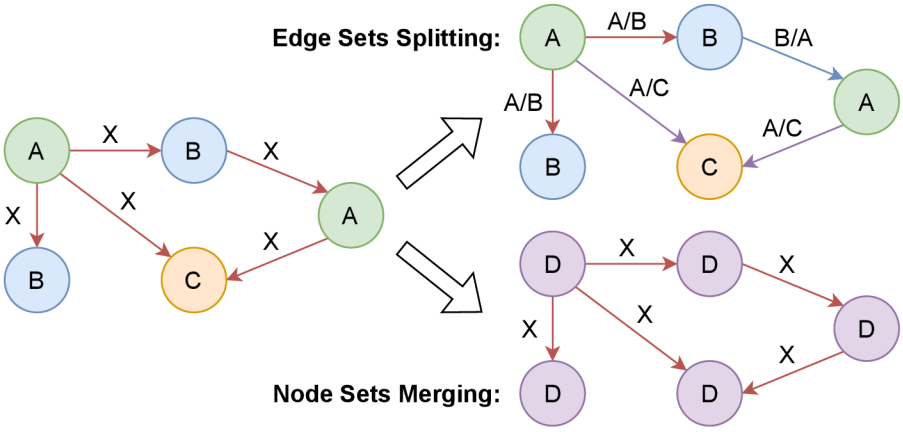

Figure 4.8: The figure demonstrates two basic ways to meet one of the conditions set by the TFGNN schema – that each edge set has exactly one source and target node set.

necessary attributes were retained during the feature selection phase. The principle is also demonstrated in Figure 4.8.

Merging of Node Sets The condition is thus ensured in this thesis by merging certain node sets. Although efforts were made in the feature selection phase to ensure that all node sets to be merged in the future have the same attributes, some attributes are too important to discard. The problem of different attribute sets when merging node sets can be solved in two extreme approaches:

1. *Sparse nodes* – create a set of all attributes contained in the merged node sets. The output super node set will have all these attributes. If an attribute does not make sense for a particular node, it is simply replaced with an invalid value. This principle is simple but creates sparse nodes and effectively increases the graph's data size. The principle of merging node sets using sparse nodes is demonstrated in Figure 4.9.

2. Latent nodes – the exact opposite approach is extracting each node's attributes into a special *data node* connected to the original node by a special edge set, according to the original node set. This again effectively unifies the node format. This principle is somewhat more complex because it requires the creation of new edge sets connecting latent nodes with their data nodes. The number of these new edge sets is $|N|$, where N is the set of merged original node sets. However, the output is a graph that is smaller in data size but larger in the number of nodes. Another advantage is that data nodes will (when using oriented GNNs) constantly send information about the original data, as their values will not be overwritten during GNN computation because they have no incoming edges. The principle of merging node sets using latent nodes is demonstrated in Figure 4.10.

Both approaches have their advantages and disadvantages. In this work, the *mixed nodes* approach is used – a combination of the best properties of both approaches. For attributes that are common to all/most original node sets (e.g., `ORDER` for all nodes inheriting from `AST_NODE`), the sparse node approach is used. On the other hand, for attributes that are

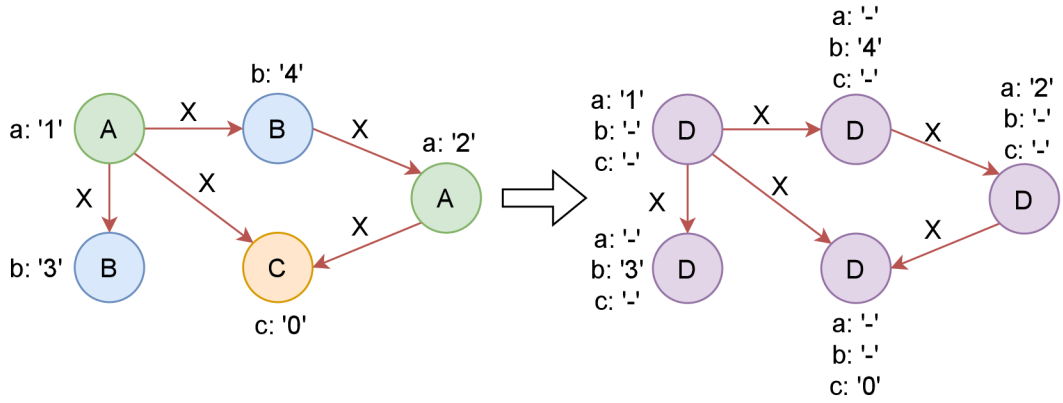

Figure 4.9: The figure demonstrates merging node sets using sparse nodes – the super node set contains attributes of all original node sets. The original graph is from Figure 4.8 and is supplemented with node set attributes.

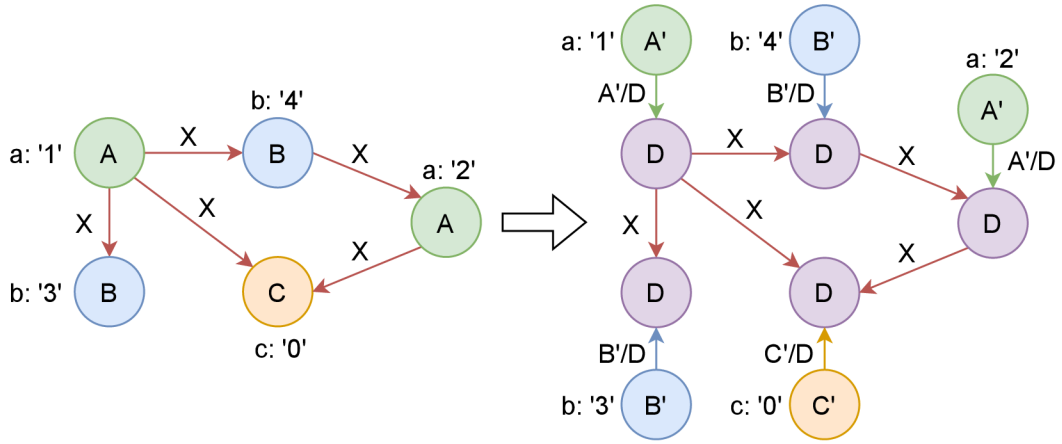

Figure 4.10: The figure demonstrates merging node sets using latent nodes – the attributes of the original node sets are extracted into special data node sets. The original graph is from Figure 4.8 and is supplemented with node set attributes.

specific to certain node sets (e.g., `CODE` for `LITERAL`, containing the literal value), the latent node approach is used. Here, however, the latent node is not empty but contains previously defined sparse attributes. By combining these methods, the graphs are small in both node count and attribute count, while only requiring the definition of a few new edge and node sets to connect data nodes with specific attributes. Table C.3 shows all selected node sets and attributes in the feature selection phase and their combination into new node sets.

The New `AST_NODE` Node Set The new node set `AST_NODE` consists of original node sets that are connected by `AST` edges. All these node sets inherit from the template node set `AST_NODE` (hence the same name), see Figure 4.4. These original node sets thus represent the code as such – the AST. Thanks to the new `AST_NODE` set, it is possible to keep the `AST`, `CFG`, `CDG`, `REF`, and `ARGUMENT` edge sets intact – their target and source node sets will be the new `AST_NODE` node set. The `AST_NODE` node set contains sparse attributes:

- `LABEL` – indicates the original node set (e.g., `BLOCK`, `LOCAL`, `METHOD`, etc.); each node contains it, so there is no need to fill it with invalid values.
- `ORDER` – is also present in all nodes.
- `ARGUMENT_INDEX` – for the original node sets `METHOD`, `LOCAL`, `METHOD_RETURN`, and `METHOD_PARAMETER_IN`, this information needs to be filled with zeros, see Table C.3. However, `ARGUMENT_INDEX` will be completely removed later, as explained below.

New Data Node Sets From Table C.4, it is evident that the original node sets `METHOD` and `LITERAL` contain special attributes that require the creation of data nodes. For the `METHOD` node set, the attributes `FULL_NAME` and `IS_EXTERNAL` need to be separated. The new data node set for the `METHOD` node set is named `METHOD_INFO` and is connected using the `METHOD_INFO_LINK` edge set, where the source is `METHOD_INFO` and the target is the new `AST_NODE` node set (original `METHOD` nodes). Similarly, for `LITERAL`, the `CODE` attribute needs to be separated into the `LITERAL_VALUE` node set and connected using the `LITERAL_VALUE_LINK` edge set, with the source being `LITERAL_VALUE` and the target being the new `AST_NODE` (original `LITERAL` nodes).

Retained Node Sets Node sets `MEMBER` and `TYPE` are retained. However, due to the creation of the new `AST_NODE` from which `EVAL_TYPE` edges originate, and the retention of `MEMBER` – which also has `EVAL_TYPE` edges – it is necessary to split the `EVAL_TYPE` edge set. The name `EVAL_TYPE` is retained for edges originating from `AST_NODE` and ending in `TYPE`. The new edge set `EVAL_MEMBER_TYPE` represents the remaining edges from `MEMBER` to `TYPE`. The reason why `MEMBER` and `TYPE` are kept in separate node sets is that they do not represent the code itself (description of computation) but provide additional information about types – thus, the node sets are logically separated.

The New `ARGUMENT_INDEX` Edge Attribute The penultimate adjustment is the transfer of the `ARGUMENT_INDEX` attribute from the new node set `AST_NODE` to the `ARGUMENT` edge set. However, this is not done for all nodes that have `ARGUMENT_INDEX`, but only for those that are the target node for some `ARGUMENT` edge. At this stage, only `ARGUMENT` edges originating from the original `CALL` node set remain. The TFGNN schema allows attributes for both nodes and edges, and therefore, this transformation saves a considerable amount of data and reduces the complexity of the `AST_NODE` nodes.

Orientation of Edges The final step is to correctly orient the edges in case oriented GNNs are used. The orientation of edge sets at this stage is shown in Table 4.1, where the edge sets that need to be reversed are highlighted in red. The reasons are:

- `ARGUMENT` – information about arguments will flow towards the `CALL` node, from which it will then propagate to the respective function.
- `EVAL_TYPE` – information about types will propagate to the nodes where it will be used – it makes no sense to propagate information from `AST_NODE` to be concentrated in `TYPE` nodes.

Table 4.1: The table contains source and target node sets for each used edge set (after feature selection). Edge sets highlighted in red have an incorrect orientation and will be reversed.

Edge Set	Source Node Set	Target Node Set
METHOD_INFO_LINK	METHOD_INFO	AST_NODE
EVAL_MEMBER_TYPE	TYPE	MEMBER
CONSISTS_OF	MEMBER	TYPE
AST	AST_NODE	AST_NODE
LITERAL_VALUE_LINK	LITERAL_VALUE	AST_NODE
ARGUMENT	AST_NODE	AST_NODE
CALL	AST_NODE	AST_NODE
CFG	AST_NODE	AST_NODE
CDG	AST_NODE	AST_NODE
EVAL_TYPE	TYPE	AST_NODE
REF	AST_NODE	AST_NODE

- `EVAL_MEMBER_TYPE` – analogous to `EVAL_TYPE`.
- `CONSISTS_OF` – information about individual `MEMBER` nodes will flow into the structure so that the structure node contains information about its members.
- `REF` – here, information will propagate from `LOCAL` nodes to identifiers so that they know it is the same variable (different `LOCAL` nodes in the same function can be distinguished using the `ORDER` attribute). For `METHOD_REF`, information about the given method will flow into that node – the function is not called here, but it is desirable to attach information to it (reversed compared to `CALL` edges).

The orientation of the other edge sets is preserved for the following reasons:

- `METHOD_INFO_LINK` and `LITERAL_VALUE_LINK` – they originate from data nodes to latent nodes, allowing data information to spread further into the graph.
- `CFG` – their direction reflects the program's control flow and the chronological order of node traversal, where the node order plays a crucial role in the manifestation of errors.
- `AST` – reversing the edges would mean that it is no longer a tree, but this is not a problem. From a message-passing perspective, the tree has an interesting property: information is copied down the tree – parents send the same message to their children, and children have only a single parent. If the `AST` edges were reversed, information would flow to the original root node, and information from the children would need to be combined in some way (the term „*pooling*“ is used, see Section 4.1.5), leading to an irreversible loss of information. However, it would be possible to read the final state from the root, where information from the entire graph would accumulate – but the graph would have to be shallow enough for information to travel from all leaf nodes to the root (because, as mentioned in Section 4.1.5, the number of message passes is a hyperparameter of the model). Pooling still occurs in GNNs because nodes in

ECPGs can have multiple incoming and outgoing edges. Thus, it might be interesting trying the reverse direction of the `AST` edges in the future.

- `CALL` – similar reason as `CFG` – it is a natural control flow of the code.
- `CDG` – edges originate from nodes representing conditions to nodes affected by the condition. Thus, it is again in the correct chronological order.

Atribute Transformation

Another property that graphs must satisfy according to the TFGNN schema is that an attribute (from now on, referred to as a feature) has a fixed type [77]. However, this is not yet the case; for example, the feature `CODE` in the `LITERAL` node can contain values of all data types in LLVM IR. Although all values can be considered strings in CSV, and TFGNN supports features of type `DT_STRING` [77], it would be more challenging for the model to extract useful information from such complex features. To facilitate training, some complex features are decomposed into multiple simpler ones [74].

Features of the `AST_NODE` Node Set The node set `AST_NODE` has a feature `ORDER`, which is a simple integer type. The feature `LABEL` is also a simple type, with categorical values representing the names of the original node sets. To prevent the model from having to take a string as input, the feature label values are mapped to a simple integer type as follows: `UNKNOWN: 0`, `METHOD: 1`, ... ,`RETURN: 11`. The model does not need the `LABEL` feature in string format; it just needs to distinguish between different types, and the simplest representation is an integer.

Features of the `MEMBER` Node Set The node set `MEMBER` has only one feature, `ORDER`, which does not require any modification.

Features of the `METHOD_INFO` Node Set The node set `METHOD_INFO` contains the flag feature `IS_EXTERNAL`, which contains values `True` and `False` since it is a flag. These boolean values are converted to integer values `1` and `0` for simplicity. The second feature is `FULL_NAME`, which contains the name of the function. Since LLVM IR operators are modeled as functions in ECPGs, `FULL_NAME` can include the prefix `<operator>.` followed by the operator's name, such as `notEquals`, `xor`, etc. Since operators are used much more frequently than functions and are also limited in number, it makes sense to convert them into their own categorical feature, `OPERATOR`. This feature contains the numerical designation of the operator if detected from the `FULL_NAME` feature. If it is not an operator, the entire function name should be remembered. Here, there are several options for modeling the remaining values:

- Keep the name as a string – the model will need to contain, for example, some kind of RNN layer.
- Use word2vec [61] or a similar model that can encode a word into a vector while preserving its meaning.

- Use trainable embedded tables [82].
- Hash the name.

In this work, the simplest and fastest approach, hashing, is used. The function name is hashed into 24 bits (the reason for 24 bits is explained below). This approach discards all information about the original name but allows the model to remember the occurrence of specific functions – if a function frequently appears in the true positive class, it is likely associated with it and can serve as part of a learned pattern. Hashing was chosen because, compared to other methods, it is by far the fastest, and generating Graph D2A (and applying feature engineering) is already computationally expensive, as described in Chapter 5. However, future work should include experiments with other string encoding methods.

Features of the `TYPE` Node Set For the node set `TYPE`, it is again necessary to decompose the feature `FULL_NAME` into simpler features that can be better processed by the model. `FULL_NAME` contains the name of the represented data type. LLVM IR supports a number of data types [55] (here are the formats as they appear in the `FULL_NAME` feature):

- Pointer – the suffix contains one or more `*` characters, e.g., `i32*`, `FILE*`, etc.
- Array – the format is `[ LEN x TYPE ]`, e.g., `[114 x i8]`, `[114 x [114 x i8]]`, etc.
- Integer – the format is `iN`, where `N` is an integer > 1 indicating the size of the type in bits, e.g., `i1` (boolean), `i32`, `i128`, etc.
- Floating point – one of `half`, `float`, `double`, or `fp128`.
- Structs – the format is `{ TYPE1, TYPE2, ... }`, e.g., `{ i32, i32 }`, `{ i1, float, { i32, i32 } }`, etc.
- Function signature – the format is `TYPE (TYPE1, ...)`, e.g., `i1 (i1, i8*)`, etc.
- Void – represents an empty value, denoted as `void`.
- Named type – everything else, e.g., `ngx_radix_tree_t`, `FILE`, etc.

In the previous section, it was described that `TYPE` nodes with signatures were removed, so they do not need to be considered. Similarly, `TYPE` nodes with the value `ANY` were removed – this is not an LLVM IR type but a value inserted by the Joern tool. The other data type names need to be appropriately represented using simpler data types. Thus, the feature `FULL_NAME` is decomposed into the following primitive features:

- `PTR` – if the type is a pointer, this stores the pointer depth – the number of trailing `*` symbols.
- `LEN` – if the type is an array, this stores its length (only the outer-most array is considered).
- `INT` – if the type is an integer, this stores `N`, the number of bits.
- `FP` – if the type is a floating point, this stores a categorical numerical designation of the given type.

- `HASH` – if the type is a struct or a named type, this stores the 24-bit hash of its name. In the case of an array, it stores the 24-bit hash of the element type name (if it is not a primitive type, see below).

The individual features are set as follows and **in exactly this order**:

1. All features are initialized to 0.
2. `PTR` is set (it can also be 0), the trailing `*` are removed, and **processing continues**.
3. `LEN` is set (it can also be 0), and if `LEN` > 0, the outer-most array is removed, leaving only the type of the elements, and **processing continues**.
4. If the type is an integer, `INT` is set, and processing ends.
5. If the type is a floating point or void, `FP` is set (0 in the case of void), and processing ends.
6. The remaining type is an array, struct, or named type – `HASH` is set (from the remaining name), and processing ends.

From this, it follows that for the void type, all values are equal to 0, which semantically makes sense because it indicates the absence of a value.

Features of the `LITERAL_VALUE` Node Set For the node set `LITERAL_VALUE`, the feature `CODE`, which contains the literal value, needs to be decomposed. It can take on all types described above, so it must be decomposed while preserving the highest possible accuracy. The primitive features will be:

1. `INT` – if the literal is an integer, this is its value.
2. `FP_MANTISSA` and `FP_EXPONENT` – if the literal is a floating point, this stores its mantissa and exponent, respectively.
3. `INVALID_POINTER` – a flag, if the type is a pointer and contains the special value `nullptr`.
4. `ZERO_INITIALIZED` – a flag, if the special value `zero initialized` is present.
5. `UNDEF` – a flag, if the special value `undef` is present.
6. `HASH` – the hashed value of arrays, structs, named types, and function pointer values (in this case, their code) into 24 bits.

It is not necessary to store detailed information about the literal type here because the data node `LITERAL_VALUE` is directly connected to the latent node `LITERAL` (`AST_NODE`), which is connected to its `TYPE` node.

Normalization of Features All features are further normalized. Normalization is a commonly used technique in machine learning that can accelerate learning and improve model performance [43, 1], especially for datasets where features have different ranges. In this thesis, simple *MinMax normalization* to the interval $< 0, 1 >$ is used. The advantage of MinMax is that it preserves the order and is very simple. Its disadvantage lies in outliers in the original data, which can cause common values to be compressed into a relatively small interval, making it challenging for the model to distinguish them. Additionally, it would be worthwhile to experiment with other normalization techniques.

Flag features `IS_EXTERNAL`, `UNDEF`, `INVALID_POINTER`, and `ZERO_INITIALIZED`, do not need to be normalized because they only contain the values `0` and `1`.

The categorical feature `OPERATOR` is divided by the number of possible values since the value `0` is reserved for an empty feature. The number of possible values is determined from the training data.

The categorical feature `LABEL` is divided by `number of possible values - 1`, where there are 12 possible labels (original node sets). Here, `0` is not reserved because `LABEL` cannot have invalid values. Similarly, the categorical feature `FP` is divided by 4 because the possible values are `void`, `half`, `float`, `double`, and `fp128`, with `0` reserved for `void` [55].

Numeric features such as `INT` (node set `TYPE`), `PTR`, `LEN`, `ORDER` (node set `AST_NODE`), `ARGUMENT_INDEX` (edge set `ARGUMENT`), and `ORDER` (node set `MEMBER`) are divided by the maximum values found among the training data (more info below).

The feature `HASH` (for all node sets) is normalized using the value $2^{24} - 1$, where 24 is the hash length in bits. The reason for 24 bits is that the `float32` type has a mantissa of 24 bits (23 bits and 1 implicit bit), according to the IEEE 754 standard [41]. It is thus possible to store a normalized number (though in the interval $< 1, 2)$, see below) of 24 bits in `float32` without loss of information. The `float32` type must be used due to the reasons mentioned in the TFGNN schema description (see below).

For the feature `INT` (node set `LITERAL_VALUE`), normalization is similar to `HASH`, except that accuracy of high values is sacrificed for better accuracy of lower values. The reason is that lower constant values are more likely to appear in control structures, such as loop counts, flags, etc., than higher values (as evidenced by checking many Infer reports in [3]). Thus, the `INT` feature is essentially truncated to `int16`, converted to unsigned, and normalized using $2^{16} - 1$ (`MAX_UINT_16`).

For the features `FP_MANTISSA` and `FP_EXPONENT`, simply dividing by the highest value is not possible due to differing magnitudes, which could result in a significant loss of information. The normalization used is based on the IEEE 754 [41] format for `float32` (which must be used, see below). The mantissa in this format is already normalized to the range $(-2, -1 >$ for negative numbers and $< 1, 2)$ for positive numbers. These intervals are only shifted to form a uniform interval $(0, 2)$ and then divided by 2 (here is a potentional loss of information). `FP_EXPONENT` can take values from 0 to 255 for `float32` (or -127 to 128 due to the implicit offset), so dividing by 255 is sufficient to normalize it in unsigned format. However, since higher than `float32` values can also appear in the graph and must be encoded in `float32` and then normalized to $< 0, 1 >$, information loss will undoubtedly occur, such that all larger floating point types are converted to `float32`. By splitting the original feature value in `float32` into two features, `FP_MANTISSA` and `FP_EXPONENT`, also

in `float32`, the encoding and normalization process will not result in a significant loss of information (only in the form of inaccurate operations).

All normalization values that need to be obtained from the dataset must be derived from the training data. If they were obtained from the test or validation sets as well, information would be transferred to the training process. Thus, model evaluation would not be accurate – model generalization would be affected to some extent. The extraction principle and sample values of the normalization coefficients are in Section 5.5.

TFGNN Schema

As previously mentioned, when using the TFGNN library, it is necessary to define the TFGNN schema [77]. It is an accurate and detailed description of the structure of heterogeneous multigraphs. The TFGNN schema designed in this thesis defines ECPGs (after feature engineering) just as Section 2.4 describes CPG graphs. However, the description here is stored in the form of Protocol Buffers [31] (often referred to as Protobuf), which are language-neutral, platform-neutral, extensible mechanisms for serializing structured data. The TFGNN schema specifically uses a protocol named `tfgnn.GraphSchema`[8]. The TFGNN schema contains information about individual *graph pieces*:

- **Context** – a set of features that apply to the graph as a whole, such as the type of Infer error.
- **Node sets** – disjoint sets of nodes where all nodes within a node set have the same set of features.
- **Edge sets** – disjoint sets of edges where all edges within an edge set have the same set of features and also share the same source node set and target node set.

Each feature definition (for context, node set, and edge set) contains the following information [77]:

- **Name** – must be unique within the graph piece.
- Description (optional).
- **Data type**:
  - Integers `DT_<INT|UINT><8|16|32|64>` or `DT_BOOL` – all stored as `int64`.
  - Floating point `DT_<FLOAT|DOUBLE|HALF|BFLOAT16>` – all stored as `float32`.
  - `DT_STRING` – stored as `bytes`.
- **Shape** – e.g., `[64]` for a vector of length 64, `-1` for ragged dimension [81], or it can be omitted, and then it is a scalar (all features in this thesis are scalars).

The schema allows specifying integer types as inputs for GNNs. However, since TFGNN model weights are in `float32`, all integer features are converted to `float32` after the first operation. These conversions are often done beforehand for type unification and better parallelization on the GPU[9]. For this reason, all features (except `label`, see below) are defined

[8] Sources of `tfgnn.GraphSchema`: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/proto/graph_schema.proto.

[9] **Graphics Processing Unit (GPU)**.

as `float32`. This final conversion of all types to `float32` is why the feature transformations in the previous text revolved around `float32`, striving to preserve accuracy for `float32`.

The last step in creating the graphs is to include the `label` and `LINE` features in the TFGNN schema context. `label` indicates whether a graph belongs to the true positive or false positive class and has the type `DT_INT32`. Including the graph label in the context is directly recommended by the TFGNN documentation [77]. The feature `LINE` contains the line number on which the error was reported by Infer, normalized by the highest such value in the training data. According to [68], this feature was the most important for Random Forest models and is thus included here as well. Again, adding other features to the context, which were used in [94, 68], could also be beneficial.

This graph context and ECPGs together form the graphs described in the TFGNN schema. These complete graphs are stored in the TFRecords format, as advised by the TFGNN documentation [77], which can be easily read as input to GNNs and are also more space-efficient than raw ECPGs (Graph D2A) – mainly due to feature engineering (see Section 5.6). The created TFGNN schema is included on the storage medium (see Appendix A) and is also available in the repository on GitHub[10].

4.1.5 Graph Neural Networks Model

The principle of GNNs (or GNN layers) was described in Section 2.3. This section further describes, in more practical terms, the architecture used in this thesis (specific models with hyperparameters are provided in Chapter 6), created in TFGNN to rank graphs (errors found by Infer) by the likelihood of being a real error. The trained model is the final output of the training pipeline, as seen in Figure 4.1. The process of models training takes graphs (with labels inside) in the TFRecords format and a TFGNN schema as inputs. The process is then composed of the following steps:

1. Load the graphs.
2. Balance the class data.
3. Create a preprocessing model.
4. Create a model, consisting of the following main parts:
   (a) A layer for initializing *hidden states* (i.e., embedding vectors from Section 2.3).
   (b) GNN layers.
   (c) *GNN head*.
5. Create a training loop.
6. Save the model.

Loading the graphs is very simple with the TFGNN support of the TFRecords format and is described in Section 5.7. The loaded data is heavily unbalanced, as shown in Tables C.1 and C.2. Even the most balanced project, libtiff, has a true positive:false positive ratio

[10]Created **TFGNN schema**: https://github.com/TomasBeranek/but-masters-thesis/blob/thesis-submission/model/schemas/extended_cpg.pbtxt.

of only about 1:20, while the least balanced project, ffmpeg, has a ratio of almost 1:140. Balancing the data is crucial because it prevents models from favoring the majority class [38] and thus helps the model learn truly useful patterns. Balancing can be done in several ways:

- Up-sampling the minority class – replicate the minority elements (in this case, true positives) so that the ratio is approximately 1:1. The elements can be replicated in several ways:
  - Duplication – does not bring any new information.
  - SMOTE[11] – create new synthetic samples from existing ones; this is a form of data augmentation [6].
- Down-sampling the majority class – randomly remove samples from the majority class until the data is balanced, but useful data are lost.
- Class weights – add weights to classes that influence learning.
- K-fold cross-validation with a split of the majority class – a type of K-validation [7], where each split will contain the same set of all minority samples, but the majority data will be divided into as many equal parts as needed so that the ratio of each part is again approximately 1:1. For an original ratio of 1:20, this would be 20-fold cross-validation. One model is trained on each split – 20 models in this case.

In this thesis, only up-sampling of the minority class is used. However, it would be beneficial to try other methods, especially SMOTE and K-fold cross-validation with subsequent voting by individual models.

The TFGNN documentation recommends creating a preprocessing model [78], which should adjust the data into its final form – feature selection, feature splitting, normalization, etc. All these tasks have already been performed earlier (see Section 4.1.4), so the preprocessing model is used only to extract the label from the graph.

The main step is to create the actual model. The models used in this thesis have a very similar architecture and are heavily inspired by the TFGNN documentation [83, 82, 80] and the examples in the TFGNN repository[12]. A slightly generalized architecture of the models used in this thesis is shown in Figure 4.11. The first part is a layer (or more) that initializes the hidden state of all nodes. The type of these layers is not strictly defined and depends on the format of the input nodes. This layer can be any differentiable function. In this thesis, the data in the nodes are a set of scalars, so only classic Dense layers (i.e., *densely-connected neural network layers*) are used (one shared layer for each node set). If the features in the nodes were, for example, ragged, RNNs would probably be used. If the data were images, CNNs could be used, etc. Or this layer can be omitted, and the first GNN layer can be used to create hidden states.

After the layers initializing the hidden state, the GNN layers follow. In this thesis, *MtAlbis layers* [80, 85] are used, which are recommended for initial experiments by the TFGNN documentation. MtAlbis layers proved to be so effective in the experiments that they were retained, see Chapter 6. These layers generalize VanillaMPNN[13] described in [29]. MtAlbis

[11] **Synthetic Minority Oversampling Technique (SMOTE)**.
[12] **TFGNN**'s repository: https://github.com/tensorflow/gnn.
[13] Vanilla Message Passing Neural Network (VanillaMPNN).

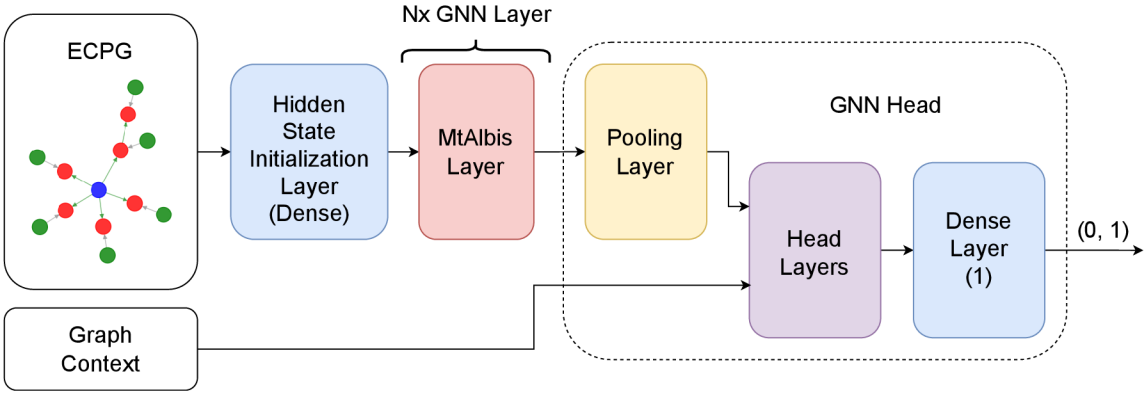

Figure 4.11: A generalized architecture of the GNN models used in this thesis. The architecture is based on the MtAlbis GNN layers and is a binary classification model – it ends with a Dense layer with a single output neuron.

work on heterogeneous graphs – which are ECPGs. One cycle of updating all hidden node states, i.e., one round of the message-passing algorithm (see Section 2.3), is a single MtAlbis layer [80]. Thus, the number of updates/layers is the depth of the model and one of the most important hyperparameters. The deeper the model, the further information from a particular node can propagate, but the more complex the model – it has more parameters, takes longer to learn, overfits more, etc.

The final part is the GNN head, which serves as the equivalent of the super node described in Section 2.3. Information from all (or only some) node sets is input into a pooling layer, whose output is a combined hidden state (as it would be for a super node). This hidden state, together with features from the graph context, is input into additional fully connected layers (or a single layer), whose output is the model's output – in this case, a single number. The model is supposed to distinguish between two classes, making it a binary classification. Thus, the last Dense layer must have a single neuron, as shown in Figure 4.11. The sigmoid function is applied to the output of this neuron (though it is not necessarily required), which converts the input number from the interval $(-\infty, +\infty)$ to $(0, 1)$. Therefore, the model's output is a single number in this interval, which is higher the more confident the model is in class 1 (true positive). Models for binary classification are most often trained using *Binary Cross Entropy* [72], as is the case with all models in this thesis.

Creating the training loop and saving models is again straightforward thanks to the use of TFGNN. Of course, it is also necessary to fine-tune hyperparameters such as learning rate, optimizer type, batch size, etc. More details can be found in Chapter 6.

Models created using the training pipeline have interesting properties:

- Thanks to the use of LLVM IR, they are **language-independent**.
- Since the output of Infer is only used for program slicing (and obtaining the type of error), it can **easily be adapted to another static analyzer**.
- It is also possible to use them **without a static analyzer**, if slicing criteria are created (possibly automatically) and the type of error is specified. This way, the

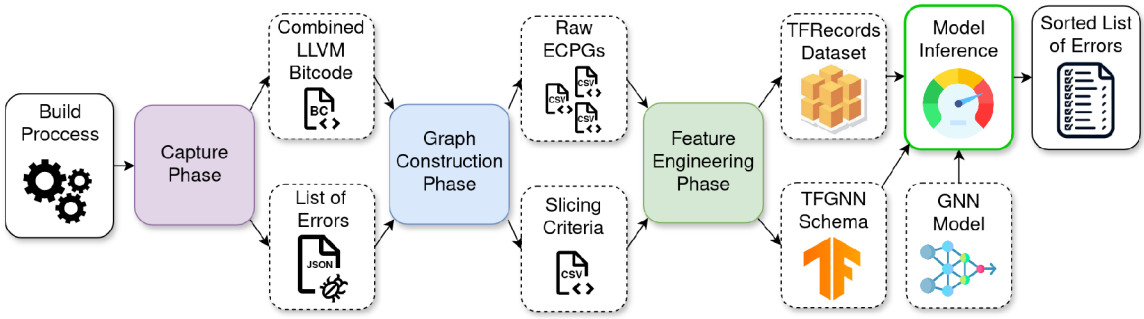

Figure 4.12: The figure shows a schematic of the inference pipeline, which ranks Infer reports on real C/C++ projects based on the probability of being true positives. The dashed boxes indicate the intermediate products and generated data. The green outline indicates tools created as part of this thesis. Icons were taken from [104, 66, 106].

models can completely replace static analysis and be used directly to find errors in the source code.

4.2 Inference Pipeline

The goal of the inference pipeline is to run Infer on a real C/C++ project, generate ECPGs for each Infer report, and then rank the reports based on their probability of being true positives using the created GNN models. As shown in Figure 4.12, the inference pipeline is fundamentally the same as the training pipeline. It differs only in the way LLVM bitcode is extracted and at the end, where models are not trained but are used solely for inference.

4.2.1 Capture Phase

The goal of the capture phase (see Figure 4.13) is to connect to a running build process and capture the information required for the graph construction phase (described in Section 4.1.2). This necessary information includes the source files compiled to LLVM bitcode and the same source files captured in Infer's capture phase (see Section 2.2), which will then be analyzed by Infer. To obtain this information, compilation commands must be extracted from the build process. This can be done by:

1. parsing the build scripts,
2. capturing the commands using a compiler wrapper.

Parsing build scripts is very challenging because each build system uses a different syntax and different techniques. However, some build systems have built-in functionality for extracting compilation commands, such as CMake [45]. Unfortunately, very few build systems have this feature. Another problem is that some software does not use any standard build systems. Instead, they use custom scripts (e.g., bash) for compilation, linking, etc. These scripts can have any structure and hierarchy of calling other scripts or tools, making it almost impossible to statically parse compilation commands from them. The use of such

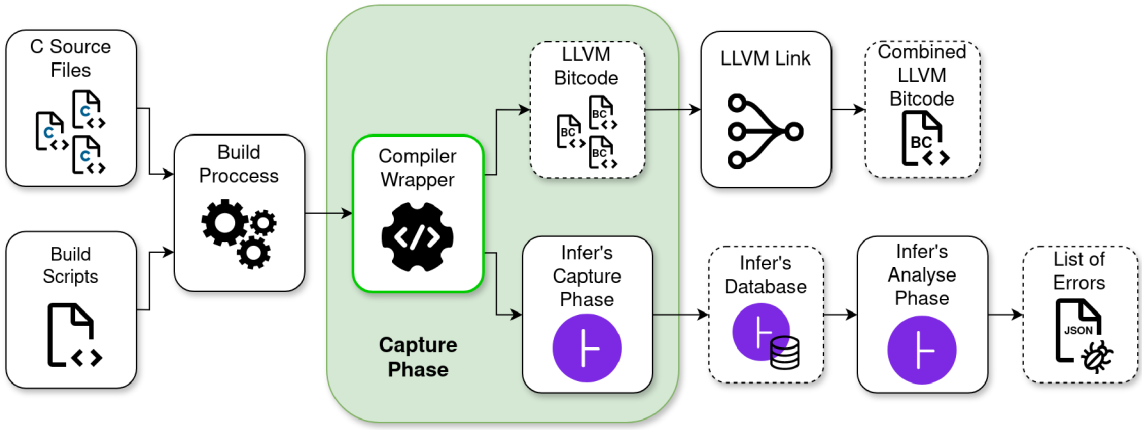

Figure 4.13: The figure shows a schematic of the capture phase, which generates LLVM bitcode and runs Infer analysis on real C/C++ software. The dashed boxes indicate the intermediate products and generated data. The green outline indicates tools created as a part of this thesis. Icons were taken from [98, 25].

scripts is relatively common in SRPM[14] packages, as found in author's previous work [3]. This thesis aims for later deployment specifically on SRPM packages and must take this feature into account. For the reasons mentioned above, parsing build scripts of unknown software is generally impractical.

The second and practically applicable option is to create wrappers over C/C++ compilers and intercept the compilation commands during the build process. The design and implementation were addressed in the author's previous work [3], so the principles of the wrappers will be described here very briefly. Every time a compiler is invoked by the build system, the installed wrapper is called. For each such call, the wrapper captures its arguments and performs the following steps:

1. It filters out options that are incompatible with Infer's internal Clang compiler, which is used to compile source files into SIL (see Section 2.2).
2. It invokes Infer's capture phase, passing the modified compilation command. Infer then stores the captured source files in SIL representation in its database.
3. It calls the original, unmodified command with the original compiler so that the build can proceed without issues.

The wrapper is designed so that even if Infer's capture phase fails, the original command is still executed. Failure of Infer's capture phase will not crash the entire analysis/pipeline, but it may increase the likelihood of generating false positives/false negatives. This error recovery is possible due to Infer's properties: if it does not have the required implementations of the analyzed functions captured, it assumes they may return any value (limited by their return type, of course). This speculation introduces a certain degree of over-approximation and thus the potential for false positives. False negatives can occur because files not captured by the Infer capture phase are not analyzed. An important note is that these compiler

[14] **Source Red Hat Package Manager (SRPM) package** – provides the source code of software via the RPM package manager for operating systems like RHEL, Fedora, and CentOS.

wrappers can (and typically are) called in parallel. Therefore, it is necessary to be aware of possible critical sections, such as Infer's database. The description of how critical sections are protected in the wrapper can be found in [3].

For the inference pipeline, it is necessary to add additional functionality to the wrapper – generating LLVM bitcode from each captured compilation command. The principle, including an example of how to generate LLVM bitcode using the compilation command, was already presented in Section 4.1.1. Unlike the training pipeline, the inference pipeline must consider input compilation commands in all possible formats, so it is necessary to remove the `-o` option (and its value) and also ensure that it is truly a compilation command (it must contain the `-c` option).

The final task that the wrapper needs to accomplish is finding all the generated LLVM bitcode files. Again, there are several ways to obtain this list of files. For instance, one could analyze compilation commands and extract the names of the compiled files or insert `-o` options. However, the simplest method is currently used here: upon the wrapper's first invocation, a list of all existing `.bc` files in the filesystem is created. After the build is finished, this process is repeated. By comparing these two lists, it is possible to identify which `.bc` files were added during the build and thus contain the LLVM bitcode.

It may seem that running multiple builds concurrently could result in `.bc` files unrelated to the current project being compiled. This issue indeed occurs with all the methods mentioned, as it is not possible to distinguish which build the `.bc` files originated from. Similarly, Infer cannot distinguish between individual projects, so it is necessary to ensure that only one project's compilation is executed at any given time. However, running multiple projects concurrently will not cause errors but will merely lead to Infer reporting errors (and generating graphs) for all the projects being compiled.

After the build and before the graph construction phase begins, two additional steps must be taken. First, **all** generated LLVM bitcode files need to be merged into a single file using the llvm-link tool. Then, the Infer analysis is performed on the captured files. After the analysis is complete, Infer generates a list of potential errors, from which slicing criteria are extracted for LLVM-Slicer during the graph construction phase. Unlike the training pipeline, where each error detected by Infer (or D2A sample in the training pipeline) generates its own LLVM bitcode file, here, a single file contains the entire source code. The LLVM bitcode files are differentiated only after slicing according to the criteria of individual reports.

4.2.2 Inference Phase

After the graph construction phase, feature engineering is applied to the raw ECPGs, just like in the training pipeline (see Section 4.1.4). The only difference is that normalization coefficients already generated from D2A are used.

Next comes the inference itself, as shown in Figure 4.12. The inference using the GNN model evaluates the input graphs – representing individual errors found by Infer – based on their probability of being true positives. The original output from Infer is then sorted in descending order according to the GNN model's score. Unsupported error types (see Section 4.1.1) and errors for which a graph could not be generated (see Section 5) are placed at the end of the list in their original relative order. Even in this sorted output, however, a typical trade-off is encountered between the number of true positives and the

number of false positives. The more true positives that are sought, the worse the true positive vs. false positive ratio becomes (to ensure that all true positives are found, all reports must still be checked). However, the mere fact that it is possible to choose this threshold is a significant advantage of these sorted outputs compared to the unsorted ones.

Chapter 5

Implementation

This chapter describes the implementation of training and inference pipelines, as designed in Chapter 4. The training pipeline consists of a series of independent tools. Specifically, Section 5.1 describes the D2A filter that removes unsupported error types. Section 5.2 discusses the implementation of a bitcode generator that creates LLVM bitcode for D2A samples. Section 5.3 describes the slicing criteria extractor. Section 5.4 details the generation of Graph D2A from LLVM bitcode. Section 5.5 explains the extraction of normalization coefficients for feature normalization. Section 5.6 outlines the implementation of feature engineering, including graph and attribute transformations and normalization. Finally, Section 5.7 and Section 5.8 cover model training and evaluation, respectively.

Unlike the training pipeline, the inference pipeline is fully automated. It comprises compiler wrappers, described in Section 5.9, and a script that automates graph creation, discussed in Section 5.10.

All source files for both the training and inference pipelines are open-source and accessible on GitHub[1]. Data manipulation primarily utilized Python, particularly libraries such as Pandas[2], NumPy[3], NetworkX[4] (nx), and TensorFlow[5]. Bash and make were used to automate the calling of scripts and other auxiliary tasks.

The following sections provide details on the computation times for various components. All measurements were conducted on Ubuntu 20.04 with the following hardware:

- CPU[6] – Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz, 6x cores, 12x threads,
- GPU[7] – NVIDIA GeForce RTX 3060 Ti, 8GB,
- RAM[8] – 16GB,
- Memory – 500GB SSD[9].

[1] All source files are available at https://github.com/TomasBeranek/but-masters-thesis.
[2] **Pandas**'s website: https://pandas.pydata.org/.
[3] **NumPy**'s website: https://numpy.org/.
[4] **NetworkX**'s website: https://networkx.org/.
[5] **TensorFlow**'s website: https://www.tensorflow.org/.
[6] **Central Processing Unit (CPU).**
[7] **Graphics Processing Unit (GPU).**
[8] **Random Access Memory (RAM).**
[9] **Solid-State Drive (SSD).**

5.1 D2A Filter

The initial step in creating the Graph D2A involves filtering out unsupported data types, as discussed in Section 4.1.1. This filtering is implemented through a script named `filter.py`, written for Python 3.8. The script takes a directory containing the D2A dataset as its input (specified using the `-d` or `--dir` option), which can be downloaded from [40]. The files, named in the format `{project}_labeler_{0|1}.pickle.gz`, correspond to individual projects and labels. Although „after-fix“ samples are available (see Chapter 3), they are not used in this thesis and are ignored by `filter.py`. The second input parameter is the output directory (specified using the `-o` or `--output-dir` option), where the results are stored (if the directory does not exist, it will be created).

Each input file in the input directory is first decompressed from the `.gz` archive using the `gzip` library. Then, using the `pickle` library, which is used for object serialization and deserialization, the samples are sequentially read. Samples belonging to the supported error types are saved in a file with the same name (including `.pickle.gz`) in the output directory. Unsupported samples are completely discarded.

From each saved sample, certain information that is no longer needed in the training pipeline is also removed. This significantly reduces the size of the samples, saving disk space and speeding up operations such as loading and saving. The removed information includes (explanations of each attribute can be found in [39]) `label_source`, `bug_loc_trace_index`, `sample_type`, `commit[changes]`, `functions`, and `zipped_bug_report`.

The command to run `filter.py` might look like this:

```
python3.8 filter.py -d d2a/ -o d2a-filtered/
```

The script removes 20,732 samples with unsupported error types, which is ~1.6 % of the total number of samples. The number of removed samples for each project is shown in Table 5.1. Filtering the entire dataset takes ~3 minutes, and the dataset size is reduced from ~3.6GB to ~288MB (calculated only with the `*_labeler_*` files).

5.2 Bitcode Generator

From the filtered D2A dataset, it is necessary to generate a single LLVM bitcode for each sample, as described in Section 4.1.1. This is accomplished using the `generate_bitcode.py` script for Python 3.8. The script must be applied individually to each `*_labeler_*` generated by `filter.py` (see Section 5.1), specified using the `-f` or `--file` option. The script also requires the directory containing the **original** project repository (using the `-r` or `--repository` option) and the project (using the `--project` option) from which the LLVM bitcode will be generated, such as `httpd`[10]). Finally, the output directory must be specified (using `-o` or `--output-dir`); if it does not exist, it will be created. Running `generate_bitcode.py` might look as follows:

```
python3.8 generate_bitcode.py -r httpd/ --project httpd \
-f d2a-filtered/httpd_labeler_1.pickle.gz -o d2a-bitcode/httpd_1/
```

[10] **httpd**'s repository: https://github.com/apache/httpd.

The script first retrieves a chronological list of all commits from the repository using:

```
git log --all --format=%H
```

Then, a set of commits for all samples from the input D2A file is obtained. From the complete list of commits, those that are not also in the set of commits from the D2A are removed – in other words, commits on which no D2A samples exist. This modified list is reversed so that the first commit is the oldest and the last is the newest. The script then iterates through individual commits (represented by their hashes) and performs the following:

1. The repository is switched to the given commit using `git reset --hard HASH`.
2. All files that are not part of the repository (especially products from previous runs) are deleted using `git clean -dfx`.
3. A project-specific set of actions required for a successful build is performed (see below).
4. For each D2A sample at this commit:
   (a) A list of files to be compiled for the given sample is obtained from D2A.
   (b) Samples consisting only of `.h` files or containing files such as `.y` or `.l` are skipped, as they do not generate LLVM bitcode when compiled.
   (c) The cache is checked to see if the LLVM bitcode for a sample with the same set of files has already been generated at the current commit. If so, a symlink[11] `{output_dir}/{sample_id}.bc` is created, pointing to the already generated LLVM bitcode. This speeds up the process because recompilation is not required and reduces memory usage since the same sample does not need to be stored multiple times. Symlinks only occupy a few bytes.
   (d) The repository is cleaned of residual files from previous compilations (at this commit) using project-specific criteria to avoid deleting essential configuration data generated when switching to this commit.
   (e) A set of already present `.bc` files in the repository is obtained; these are not LLVM bitcode files.
   (f) For each record (compiled file) in the D2A attribute `compiler_args`:
      i. Adjust the D2A compiler arguments – replace <*repo*> with the repository path and remove arguments starting with <*sys*>, which include external libraries specific to httpd (these libraries are included with their own paths, see below).
      ii. Add missing project-specific include arguments (`-I`).
      iii. Add arguments to generate LLVM bitcode (see Section 4.1.1).
      iv. Execute the generated compilation command.
   (g) Using the previously located existing `.bc` files, obtain a list of newly added files – generated LLVM bitcode files.
   (h) Ensure that the same number of LLVM bitcode files have been generated as there are original `.c` files (`.h` files are included – they do not generate separate LLVM bitcode).

[11] **Symlink** – A special type of file that points to another file in the filesystem.

(i) Use `llvm-link` to combine all LLVM bitcode files of the current sample into a single LLVM bitcode file and save it to `{output_dir}/{sample_id}.bc`.

(j) Finally, note in the cache which files were used to generate this LLVM bitcode.

After switching to the new version of the repository (new commit), a pre-compilation configuration is required. This typically involves generating platform-specific `.h` or `.c` files, generating configuration files, setting the correct paths to libraries, etc. This phase is different and quite extensive for each project, so only the procedure for httpd will be described here as an example. Configuration details of other projects can be found directly in `generate_bitcode.py`.

The httpd project is the only one that requires downloading external libraries in advance. Specifically, `apr-1.7.4` and `apr-util-1.6.3`, which can be downloaded from the Apache website[12], and the `pcre2-10.42` library, which is available in the `pcre2` repository[13]. These libraries must be renamed to `apr`, `apr-util`, and `pcre` and moved to the `httpd-dependencies/srclib` directory, which must be at the same level as the httpd repository. Furthermore, all libraries need to be configured according to their instructions (pre-configured libraries are included in the attached media; see Appendix A).

The `generate_bitcode.py` script moves to the httpd repository and prepares for the httpd compilation as follows:

1. Copies the external libraries `apr` and `apr-util` to the `srclib/` directory in the repository.
2. In some versions of the repository that contain the `pcre` library, it is necessary to initiate configuration by first running `./buildconf` (still in the root directory of the repository), which creates `srclib/pcre/configure`. Then, switch to `srclib/pcre/` and run `./configure` to generate the necessary header files for the `pcre` library, such as `config.h`.
3. If the `pcre` library is not present, the script copies the already configured one from `../httpd-dependencies/srclib/pcre/` into `srclib/`.
4. The script then checks whether any of the tracked files have changed from the last version of the project:
   - `include/ap_config_auto.h.in`,
   - `include/ap_config_layout.h.in`,
   - `modules/ssl/ssl_policies.h.in`,
   - `buildconf`.

   These are templates for the generated `.h` files and the configuration file. If none of these files have changed, the previously generated `.h` files can be reused – copy them back from `/tmp/d2a_pipeline/`. This saves a lot of time since generating them and running `./buildconf` is time-consuming across many commits.

[12] **Apache**'s website: https://apr.apache.org/download.cgi.
[13] **pcre2**'s repository: https://github.com/PCRE2Project/pcre2/releases/tag/pcre2-10.42.

5. If any template has changed, `./buildconf` and `./configure` must be rerun. The configuration process takes about 20 seconds, but it has to be done for thousands of commits.

6. The script checks another set of tracked files that are generated differently:

   - `server/gen_test_char.c`,
   - `srclib/pcre/dftables.c`.

   If they have not changed, again copy them from previous versions.

7. If they have changed, they need to be generated as follows:

   (a) `include/test_char.h` – generated using `gcc -Isrclib/apr/include -Isrclib/apr-util/include server/gen_test_char.c -o gen_test_char` followed by `./gen_test_char > include/test_char.h`.

   (b) `include/chartables.c` – generated using `gcc srclib/pcre/dftables.c -o dftables` followed by `./dftables include/chartables.c`. In some newer versions, it is necessary to check if `include/chartables.c` was created, and if not, it must be generated using `./dftables > include/chartables.c` instead.

For each version (commit) of the project, running the configuration multiple times should be avoided. To prevent losing the generated configuration files, a project-specific cleanup that preserves the contents of certain directories is used. For httpd, the command is:

```
git clean -dfx --exclude=srclib/ --exclude=include/
```

When starting the compilation for individual files of each sample, arguments extracted from D2A are used. However, most are insufficient as they do not include necessary `-I` paths for various header files. This may be due to different methods of installing libraries when creating D2A, so these paths need to be added. For httpd, `-Iinclude`, `-Isrclib/apr/include`, and `-Isrclib/apr-util/include` are appended.

As previously indicated, some samples might be skipped, or their compilation may fail. Statistics for individual projects are presented in Table 5.1. For httpd_1, the size of the filtered D2A is ~47KB, compared to the unfiltered D2A, which is ~629KB. The generated LLVM bitcode for httpd_1 is ~22MB. The script for `httpd_1` runs for ~330 seconds, which equates to ~1.6 seconds per sample. Parallelization would speed this up, but since the project repository is a critical section accessed almost continuously, it would be necessary, for instance, to duplicate it. Thus, parallelization of this process is left for potential future improvements. Similarly, enhancements to the automated project configuration are also left for future improvements, as they could improve the success rate of bitcode generation. However, compilation issues must be resolved manually, which consumes an enormous amount of time.

5.3 Slicing Criteria Extractor

To enable slicing of the generated LLVM bitcode from Section 5.2, it is first necessary to extract slicing criteria from the filtered D2A from Section 5.1. For this purpose, the Python

3.8 script `slicing_criteria_extraction.py` is provided. It takes as input a single file from the filtered D2A (specified using the `--d2a` option). The script outputs the slicing criteria in the CSV format (without header) to `stdout`, with the following columns:

1. `status` – 0 means success, 1 indicates an internal error.
2. `bug_id` – the `id` of the sample.
3. `entry` – name of the entry function (see Section 4.1.2).
4. `file` – the file where the error is located.
5. `fun` – the function where the error is located.
6. `line` – the line number where the error is located.
7. `variable` – the variable associated with the error.

An example of running `slicing_criteria_extraction.py` might look like this:

```
python3.8 slicing_criteria_extraction.py --d2a d2a-filtered/ \
httpd_labeler_1.pickle.gz > slicing-info/httpd_labeler_1.csv
```

The `slicing_criteria_extraction.py` script is used for both the training and inference pipelines. This is because both D2A and Infer's output are in the JSON format, and since D2A originates from Infer's output, they are quite similar. When the script is used on the D2A sample, it is converted to the same format as Infer's output via the simple `transform_d2a_sample` function, which essentially involves renaming and splitting some D2A attributes.

For each sample/report, a function `extract_{error_type_group}` (extracting the slicing criteria) is invoked based on its type – the 6 groups listed in Section 4.1.2. Retrieving `entry`, `file`, `fun`, and `line` is straightforward: the correct attributes are simply extracted from the JSON (see Section 4.1.2). If `variable` is extracted, it is obtained from the `qualifier` field. For `bug_id`, `id` from D2A is used in the case of D2A. For Infer, the samples are labeled incrementally starting from `0`, and unsupported sample types are skipped in the numbering to preserve the original numbering in Infer's output.

The script skips unsupported error types. If an unknown format of a supported error is encountered, the script returns `status = 1` and tries to extract at least `entry`, `file`, `fun`, and `line` from the basic information to allow slicing based on the line number.

If the `file` is a header file (`.h`), the `file` field is left empty because of future slicing – because slicing based on header files is not supported by LLVM-Slicer in the standard format. Instead, slicing should be done using only `fun` and `line`, excluding the `file` field. If the `file` contains a regular `.c` file, `fun` is omitted because `file` and `line` are sufficient to determine the slicing criteria unambiguously. Extracting slicing criteria from the filtered D2A completes for all files in under a minute.

5.4 Graph Construction Script

The bash script `construction_phase_d2a` is used for generating Graph D2A from LLVM bitcode (created in Section 5.2) and slicing criteria in the CSV format (created in Section 5.3). This script implements the remaining transformations described in Section 4.1.2. The script accepts the following position-dependent arguments:

1. The output directory for storing raw ECPGs.
2. The file containing slicing criteria.
3. The directory containing LLVM bitcode.
4. (optional) The sample number at which to end.
5. (optional) The sample number from which to start.

The `construction_phase_d2a` script can be executed with a command such as:

```
./construction_phase_d2a graph-d2a/httpd_1 httpd_labeler_1.csv \
d2a-bitcode/httpd_1
```

The script `construction_phase_d2a` operates as follows:

1. Records from the slicing information (from its copy) that already have a directory with raw ECPG are removed. This allows for the intermittent transformation of the dataset.
2. The slicing information file is divided into smaller files of 100 lines each (the last file may be smaller) and stored in `/tmp/construction_phase_d2a/split_files/`.
3. Each file in `split_files/` is then processed as follows:
   (a) The `create_cpgbin` function is called in parallel for each line in `file` using the command:

   ```
   cat ${file} | parallel --colsep ',' create_cpgbin {1} {2} {3} \
   {4} {5} {6} {7}
   ```

   This function generates a binary CPG for each line from LLVM2CPG (detailed description of this function is provided below).
   (b) If `/tmp/construction_phase_d2a/cpg/${bug_id}.cpg.bin.zip` was not generated for some samples, these samples are removed from `file`.
   (c) A script for Joern is generated, containing commands to load and re-save all `.cpg.bin.zip` files, thereby expanding them into ECPGs. An example Joern script is provided in Listing 5.1.
   (d) Joern processes all (up to 100) ECPGs.
   (e) The `cpgbin_to_csv` function is called in parallel for each line in `file`, but only the `bug_id` column is used:

   ```
   cat ${file} | parallel --colsep ',' cpgbin_to_csv {2}
   ```

```
importCpg("/tmp/construction_phase_d2a/cpg/httpd_27 ... 17_1.cpg.bin.zip")
importCpg("/tmp/construction_phase_d2a/cpg/httpd_04 ... 1e_1.cpg.bin.zip")
// more imports
importCpg("/tmp/construction_phase_d2a/cpg/httpd_3e ... 2c_1.cpg.bin.zip")
importCpg("/tmp/construction_phase_d2a/cpg/httpd_1d ... 02_1.cpg.bin.zip")
save
```

Listing 5.1: An example of an automatically generated Joern script for `httpd_1`. The script only includes `importCpg` to load binary CPGs (up to 100) and concludes with the `save` command, which saves the graphs.

This function converts binary ECPGs to CSV – raw ECPGs, stored in the output directory. Each raw ECPG has its own directory (named after its `bug_id`) containing CSV files.

(f) Finally, all temporary files created during the current iteration are cleaned up to prevent accumulation of logs and intermediate files, which could unnecessarily consume memory.

4. Statistics on the number of successful/unsuccessful samples are then calculated and printed.

The aforementioned function `create_cpgbin`, which takes a line with slicing criteria as input, works as follows:

1. First, it checks whether the input directory with LLVM bitcode contains the bitcode for the current sample. If it does, the function continues.

2. The LLVM-Slicer is called using:

   ```
   timeout 3s llvm-slicer --sc="${file}#${fun}#${line}#${variable}" \
   --entry=${entry} -o=${bc_sliced} ${bc_combined}
   ```

   The `timeout` command ensures that `llvm-slicer` completes its run. Experiments have shown that it can sometimes get stuck or run for several minutes, which is unacceptable given the large number of samples. By removing certain columns from the slicing criteria (see Section 5.3) and leveraging the behavior of variables in bash, it is possible to call `llvm-slicer` uniformly for both `.c` and `.h` files.

3. If `llvm-slicer` is successful, a CPG in binary format is generated using `llvm2cpg` as follows:

   ```
   llvm2cpg ${bc_sliced} --output=${cpg_bin}
   ```

   where `${cpg_bin}=/tmp/construction_phase_d2a/cpg/${bug_id}.cpg.bin.zip`.

The `cpgbin_to_csv` function, which takes only `bug_id` as input, simply calls `joern-export` as follows:

```
joern-export --repr all --format neo4jcsv \
-o "${output_dir}/${bug_id}" ${joern_cpg_bin}
```

where `${joern_cpg_bin}` contains the path to the binary ECPG in the temporary directory `/tmp/construction_phase_d2a/workspace/${bug_id}.cpg.bin.zip/cpg.bin`. At the end of the function, the success of `joern-export` is checked.

As mentioned earlier, the creation of binary CPGs and the conversion of binary ECPGs to CSV are parallelized using the `parallel` tool. However, the bottleneck here is Joern, which, despite running multiple instances, does not provide any speedup. Moreover, starting up Joern takes multiple seconds, so ideally, it is best to start and stop it as little as possible, hence it works in batches of 100. Larger batch sizes have been tested to further reduce the startup load of Joern, but the following issues were found:

- Batch > 5000 – Joern crashes.
- Batch > 3000 – Joern may get stuck in an infinite loop.
- Batch > 500 – Joern non-deterministically generates incomplete graphs (missing edge sets like `CDF`, `CFG`, etc.).
- Batch $= 100$ – Joern works correctly.

The following results were measured on `httpd_1`. The non-parallelized script generates approximately 500 graphs per hour. By parallelizing both of the phases mentioned above, the script reaches approximately 1100 graphs per hour. Moving Joern to batch mode allows the script to generate approximately 4000 graphs per hour. Other projects have been found to contain, on average, larger graphs than `httpd`, so the number of graphs per hour may be smaller for those projects. The output graphs for httpd_1 are ~240MB.

As hinted earlier, some samples may fail. The largest contributing factor is the timeout for LLVM-Slicer. It is possible to increase the timeout, but that would decrease the number of graphs per hour. The number of successfully generated samples can be seen in Table 5.1.

5.5 Normalization Coefficients Extractor

Before applying feature engineering (see Section 4.1.4), it is necessary to extract normalization coefficients from individual projects in Graph D2A (created in Section 5.4). This is the task of the `find_normalization_coefficients.py` script for Python 3.8. The script is executed for each project separately with 6 position-dependent arguments:

1. directory with false positives in Graph D2A of the specific project,
2. directory with true positives in Graph D2A of the specific project,
3. project name (httpd, nginx, or libtiff),
4. `splits.csv` file providing the data split into train, val, and test sets, downloadable from [40],
5. slicing criteria for false positives of the specific project,
6. slicing criteria for true positives of the specific project.

Running the `find_normalization_coefficients.py` script for httpd could look like this:

```
python3.8 find_normalization_coefficients.py graph-d2a/httpd_0/ \
graph-d2a/httpd_1/ httpd d2a/splits.csv httpd_labeler_0.csv \
httpd_labeler_1.csv
```

The `find_normalization_coefficients.py` script works as follows:

1. It processes `splits.csv` and selects the set of `id` samples belonging to the input project and the training set.
2. It then iterates over all false positive samples and then all true positive samples (the order does not matter) as follows:
   (a) If the sample `id` is not in the training set, it is skipped – obtaining information from the validation and test sets is avoided because it could affect the experiments.
   (b) For each CSV header file (`*_header.csv`) of the current sample:
      i. If the current header file does not belong to the original node sets of the merged node set `AST_NODE` (see Section 4.1.4), the node set `TYPE`, or the node set `MEMBER`, proceed to the next header file.
      ii. The corresponding data file `*_data.csv` is read.
      iii. If the header is `nodes_TYPE_header.csv` (`TYPE` node set), `LEN` and `PTR` values are extracted (see Section 4.1.4) and if their maximum values are greater than the currently found ones, they are updated. Then, proceed to the next header file.
      iv. For a header file from the merged `AST_NODE` node set or `MEMBER` node set, the values `MEMBER_ORDER` (for node set `MEMBER`) and `ORDER` (for all others) are updated (see Section 4.1.4).
      v. If the header file is for the node set `METHOD`, newly found operators (if any) are added to the `OPERATORS` set.
      vi. If the header file has a column `ARGUMENT_INDEX`, it is stored together with the column `ID`.
   (c) From the file `edges_ARGUMENT_data.csv`, obtain the set of target nodes for `ARGUMENT` edges. From previously stored `ARGUMENT_INDEX`, discard those that are not target nodes for `ARGUMENT` edges (using `ID`s). From the remaining ones, update the maximum value of `ARGUMENT_INDEX`.
3. Extract the maximum value of `LINE` (for graph context, see Section 4.1.5) from the files with slicing criteria.
4. Finally, print all normalization coefficients.

The `find_normalization_coefficients.py` script outputs its results to `stdout` in the format shown in Listing 5.2. Normalization coefficients `ARGUMENT_INDEX`, `LEN`, `LINE`, `ORDER`, `MEMBER_ORDER` and `PTR` are the maxima of all found attributes. `OPERATORS` is the set of all found operators. And `BUG_TYPES` is the set of all supported error types (see Section 4.1.1). Extraction for the httpd project takes about ~260s.

```
{'ARGUMENT_INDEX': 14,
 'BUG_TYPES': ['NULL_DEREFERENCE',
               // more error types
               'UNINITIALIZED_VALUE'],
 'LEN': 65536,
 'LINE': 9162,
 'MEMBER_ORDER': 75,
 'OPERATORS': {'<operator>.addition',
               '<operator>.addressOf',
               // more operators
               '<operator>.subtraction',
               '<operator>.xor'},
 'ORDER': 1471,
 'PTR': 4}
```

Listing 5.2: An example of the normalization coefficients for the httpd project, generated by the `find_normalization_coefficients.py` script.

5.6 Feature Engineering Script

After extracting the normalization coefficients (described in Section 5.5), feature engineering (designed in Section 4.1.4) can be applied to Graph D2A (created in Section 5.4) to produce a dataset in the TFRecords format. All feature selection, graph transformations, and attribute transformations are implemented using the `feature_engineering.py` script for Python 3.8. The script is called separately for each project and label with 8 position-dependent arguments:

1. TFGNN schema file (designed in Section 4.1.4),
2. output file name,
3. project name (httpd, libtiff, nginx, ...),
4. label (`0` or `1`),
5. `splits.csv` file,
6. filtered D2A file (`*_labeler_*`),
7. file with slicing criteria,
8. (optional) Which of the normalization coefficients to use (`httpd`, `libtiff`, `nginx`, or `nginx+libtiff+httpd`). If the argument is missing, the project value is used.

The script reads directories with individual samples from its `stdin` (one directory per line). Running `feature_engineering.py` for `httpd_1` might look like this:

```
find graph-d2a/httpd_1 -mindepth 1 -type d | python3.8 \
feature_engineering.py extended_cpg.pbtxt \
tfrecords/httpd_1.tfrecords httpd 1 d2a/splits.csv \
d2a-filtered/httpd_labeler_1.pickle.gz httpd_labeler_1.csv
```

The `feature_engineering.py` script does the following for each input Graph D2A sample:

1. Loads only those node/edge set files that are not to be removed, with the exception of the `TYPE_DECL` node set, which is removed later (see Section 4.1.4). If any used edges were connected to a node that was not loaded, it becomes an invalid node that needs to be removed appropriately. During loading, a merged `AST_NODE` is also created, and the original node set name is stored in the `LABEL` attribute of each `AST_NODE` node.
2. Discards unused node set attributes.
3. From the loaded nodes and the `AST` node set, a `MultiDiGraph` representation is created using the `nx` library optimized for graph processing.
4. Using the simple algorithm described in Section 4.1.4, all invalid nodes are removed from the graph (for now, it is just a set of ASTs).
5. Using `nx.weakly_connected_components(G)`, all WCCs are obtained, and those consisting only of `BLOCK` nodes are removed.
6. All leaf `BLOCK` nodes are also removed. At this stage, all currently present nodes are considered valid (although some will still be removed later).
7. Other edge sets are added to the graph, with the `ARGUMENT` edges only added if they originate from a `CALL`, meaning:

   ```
   G.nodes[edge['start']]['type'] == 'CALL'
   ```

8. Newly added edges may again create invalid nodes, which can now be easily removed along with their edges, as removing them will not disconnect the ASTs.
9. The graph optimizations described in Section 4.1.4 are then performed:
   (a) removing AST children of external methods,
   (b) removing unnecessary `EVAL_TYPE` edges,
   (c) removing all `TYPE_DECL` nodes,
   (d) removing unused `TYPE` nodes.
10. At this stage, it is verified that the graph forms a single WCC because no further edge or node removals will be performed that could split the graph into multiple WCCs.
11. All `METHOD` and `LITERAL` nodes are split into data and latent nodes (see Section 4.1.4), which adds the node sets `METHOD_INFO`, `LITERAL_VALUE`, and also the edge sets `METHOD_INFO_LINK` and `LITERAL_VALUE_LINK`.
12. At this stage, all node/edge sets are converted to separate `DataFrame` tables using the `Pandas` library, which is optimized for tabular operations. From this point onward, attributes of individual edge/node sets are processed in groups, not the graph structure itself.
13. All attributes are split as needed and normalized using the extracted normalization coefficients (see Section 4.1.4).

```
for edge_set_name, val in edgeset_info.items():
    source_nodeset=val['SOURCE']
    target_nodeset=val['TARGET']

    get_source_node_loc =lambda id: graph_in_dfs[source_nodeset].index.
        get_loc(id)
    get_target_node_loc =lambda id: graph_in_dfs[target_nodeset].index.
        get_loc(id)

    graph_in_dfs[edge_set_name]['source']=graph_in_dfs[edge_set_name] \
      ['source'].apply(get_source_node_loc)
    graph_in_dfs[edge_set_name]['target']=graph_in_dfs[edge_set_name] \
      ['target'].apply(get_target_node_loc)
```

Listing 5.3: An example of Python code that converts Joern node IDs into TFGNN IDs.

14. Now, it is necessary to convert `SOURCE` and `TARGET`, which contain the node IDs in all edges. Currently, nodes have IDs in ascending order starting from 1. However, TFGNN identifies nodes differently. They are numbered in ascending order starting from 0, but within node sets – meaning there can be two or more nodes with the same ID if each is in a different node set. Since edge sets must define source and target node sets (see Section 4.1.4), there will be no collisions.
15. Finally, the orientation of some edge sets is reversed (see Section 4.1.4).
16. A TFGNN `GraphTensor` is created using the `from_pieces` and `from_fields` methods [79],
17. Serializes the `GraphTensor` objects into `tfrecords` files according to whether the sample belongs to the train, val, or test set.

The outputs of the `feature_engineering.py` script are files with the `*.train`, `*.val`, and `*.test` extensions in the TFRecords format. Some samples may be faulty – for instance, no `AST` edge set was generated for them (by Joern), which must always be present in a valid sample. The number of successfully generated samples is shown in Table 5.1. The script runs on `httpd_1` for approximately ~80 seconds. The script was parallelized at the level of individual samples, but parallelization did not bring any significant speed improvement (likely because the libraries used are already internally parallelized), and some calls to the TensorFlow library (e.g., writing to TFRecords) did not work and would need to be locked into critical sections. Therefore, the parallelization was removed. The output TFRecords files for httpd_1 are ~15MB.

5.7 Model Training Script

After creating TFRecords files, training of GNN models can be done using the script `mixed_nodes_model.py` for Python 3.8. This script takes 4 position-dependent arguments:

1. TFGNN schema,

Table 5.1: The table shows the number of samples after each phase of the training pipeline. For values marked with *, the loss is not final as they were not transformed into TFRecords. However, the table indicates that this final transformation is almost lossless.

Project	D2A	Filtered D2A	Bitcode	Graph D2A	TFRecords	Loss
httpd_0	12475	11974	11818	9705	9705	22 %
httpd_1	217	210	210	193	193	11 %
nginx_0	17945	17209	17172	16741	16741	7 %
nginx_1	421	418	417	407	407	3 %
libav_0	236415	234062	226213	186614	186595	21 %
libav_1	4614	4575	4398	3331	3331	28 %
libtiff_0	12096	11385	11377	9276	9276	23 %
libtiff_1	553	534	534	459	459	17 %
openssl_0	343148	332584	301934	278292	-	20 %*
openssl_1	8022	7913	7581	6918	-	14 %*
ffmpeg_0	654891	649255	633997	500791	-	24 %*
ffmpeg_1	4826	4772	4621	3938	-	18 %*

2. directory with TFRecords,
3. output directory for saving models,
4. (optional) the value `combined` to train a single model across multiple projects (see Section 6.2); if omitted, a separate model is trained for each project.

The script `mixed_nodes_model.py` trains models on training data (and validates on validation data) of the projects httpd, libtiff, and nginx. It expects files named according to the pattern:

```
{TFRecords_dir}/{httpd|libtiff|nginx}_{0|1}.tfrecords.{train|val}
```

The script then operates as follows:

1. Data are loaded using `tf.data.TFRecordDataset` – positive and negative samples separately (validation data are loaded all at once, as shuffling is not necessary).
2. Up-sampling is applied to the minority class.
3. Positive and negative samples are interleaved.
4. All samples are shuffled to mix the samples from the individual projects.
5. Datasets are batched.
6. A `preprocessing` model is applied, which extracts the labels (see Section 4.1.5).
7. The function `train_model` is then called, performing:
   (a) First, a model is constructed using the `build_model` function, which utilizes the Keras API[14] and operates as follows:

[14] **Keras API**'s documentation: https://www.tensorflow.org/guide/keras.

i. An `Input` layer is created, taking graphs defined by the TFGNN schema as input.
ii. A `Dense` layer initializing hidden states for each node set (and optionally edge sets) is added, using `MapFeatures`.
iii. GNN layers `mt_albis.MtAlbisGraphUpdate` are added.
iv. A `Pool` layer is added.
v. `Dense` layers in the GNN head are added, combining context features and the output of the `Pool` layer.
vi. Finally, a `Dense` layer with a single output and sigmoid activation function is added.

(b) The loss function, metrics, and optimizer are set, and the model is compiled using `model.compile`.

(c) An `EarlyStopping` callback monitoring Area Under the Receiver Operating Characteristic Curve (AUROCC) (see Section 6.1) on validation data is set.

(d) Finally, the training loop is initiated using `model.fit`.

8. Thanks to the `EarlyStopping` callback, the output of the training is the model with the highest validation AUROCC found. This model (or models) is then saved to the output directory. Directories with models are automatically saved with the prefix `{ID}_`, where `ID` is a unique number – the largest found in the directory, increased by one. The directory name might look like `8_AUC_0.818`, where the AUC value specifically refers to the validation AUROCC (or their average in case of multiple models). The values of hyperparameters set in the dictionary `hyperparameters` are stored in the output folder in the file `hyperparameters.json`.

The entire model architecture is defined in the `build_model` function. Older versions of this script for earlier models can be found in the repository under commits named `Model {ID} - AUC 0.XYZ`. These historical versions, though executable, do not represent the final form of the training script and should only be used for insight into the architecture definition.

5.8 Model Evaluation Script

Trained models can be evaluated using the `evaluate_model.py` script for Python 3.8. The models are evaluated based on two metrics – AUROCC and Top N % Precision (see Section 6.3). Examples of these for the libtiff project and top-performing models are shown in Figure C.2 and Figure C.6, respectively. The script accepts 5 position-dependent arguments:

1. TFGNN schema,
2. directory with TFRecords files,
3. directory with saved models,
4. model `ID`,
5. dataset type – `test`, `val`, or `train`.

The script initially loads the data in the TFRecords format from the same location and with the same naming convention as used by `mixed_nodes_model.py` (see Section 5.7), with additional `*.test` files. No data shuffling or modifications are required for the evaluation. A preprocessing model is applied to extract labels from the graphs. Then, the model (or models) is loaded using `tf.keras.models.load_model`, and inference is performed on the data using `model.predict`. The results are provided to the `plot_top_N_precision` function, which plots the precision dependency on the number of top-selected samples. Additionally, the `plot_ROC_curve` function is called to create ROC curves. Both graphs are displayed and also saved in the current directory under the names `ROC_curves.svg` and `Top_N_precisions.svg`.

The script can be run in a special mode that creates graphs for predefined scenarios (all graphs in Chapter 6 were created using these scenarios), by passing the following special model `ID` values (4th argument):

1. `combined` – testing top performing models from Section 6.3 on combined data from the httpd, libtiff, and nginx projects.
2. `httpd` – testing top performing models on httpd.
3. `libtiff` – testing top performing models on libtiff.
4. `nginx` – testing top performing models on nginx.
5. `libav` – testing top performing models on libav – this involves cross-analysis. Files `libav_{0|1}.tfrecords.{train|test|val}` are required.
6. `chatgpt` – comparing the top performing model with ChatGPT4 (see Section 6.4). The file `libtiff-chatgpt.tfrecords.test`, containing selected samples, is needed.

5.9 Compiler Wrapper

The compiler wrapper originates from the author's previous work [3], where its implementation is also described. Here, only a brief overview will be provided, focusing mainly on its inputs and outputs for integration with other parts of the inference pipeline.

The compiler wrapper is a bash script that replaces C/C++ compilers, and the original compiler binaries are renamed to `{compiler}-original` (e.g., `gcc` becomes `gcc-original`). The repository includes a `Makefile`[15] for installing wrappers for many commonly used C/C++ compilers.

The wrapper works by intercepting all commands that would go to the original compilers, and:

1. passes them to Infer for analysis,
2. generates LLVM bitcode,
3. and finally forwards them to the original compilers.

[15] **Makefile for installing wrappers**: https://github.com/TomasBeranek/but-masters-thesis/blob/thesis-submission/inference-pipeline/Makefile.

The wrapper stores information in the `/tmp/infer-out` directory, which is generated directly by Infer. It contains the results of Infer's analysis and also a list of `.bc` files found before the first generation of LLVM bitcode. These existing `.bc` files are not generated by the wrapper (or are outdated). Their list is stored in the `/tmp/infer-out/old_bc_files.txt` file. This step needs to be optimized in future versions because it can be slow on large filesystems.

5.10 Inference Pipeline

Unlike the training pipeline, which is implemented as a series of standalone tools, the inference pipeline is fully automated. The inference pipeline uses the existing scripts `construction_phase_d2a` (see Section 5.4), `feature_engineering.py` (see Section 5.6), and compiler wrappers (see Section 5.9). The inference pipeline is a bash script named `inference_pipeline`, which combines the previously mentioned scripts and provides additional functionality, particularly data conversion into formats expected by the already created scripts. The `inference_pipeline` script should be called in the following context:

1. First, the compiler wrappers need to be installed.
2. Then, the analyzed project needs to be compiled (anywhere in the filesystem).
3. After the compilation is complete, `inference_pipeline` is called with a single parameter that specifies the output directory, for example:

   ```
   ./inference_pipeline ./
   ```

4. Finally, it is advisable to uninstall the compiler wrappers.

The `inference_pipeline` script itself works as follows:

1. First, Infer analysis is run on the `/tmp/infer-out` directory created by the compiler wrapper (see Section 5.9).
2. All `.bc` files are found in the filesystem, and those that have been added compared to the `/tmp/infer-out/old_bc_files.txt` list created by the compiler wrapper at the start of the compilation are identified.
3. Using `llvm-link`, all new LLVM bitcode files are merged into a single file named `/tmp/infer-out/combined.bc`.
4. Next, the `slicing_criteria_extraction.py` script (see Section 5.3) is executed to extract slicing criteria from the Infer output – the `/tmp/infer-out/report.json` file.
5. The `/tmp/infer-out/bitcode` directory with the LLVM bitcode must then be prepared as expected by the `construction_phase_d2a` script. Since there is only a single `combined.bc` file for the entire project, an artificial directory is created and populated with symlinks that all point to the `combined.bc`. The symlinks are named according to the IDs of individual Infer reports (see Section 5.3).
6. Now, `construction_phase_d2a` can be executed as follows:

```
../dataset/construction_phase_d2a /tmp/infer-out/raw-ecpg \
/tmp/infer-out/slicing_info.csv /tmp/infer-out/bitcode
```

7. The generated raw ECPGs are then processed by `feature_engineering.py` in inference mode – the input consists of exactly 4 position-dependent arguments: specifically, the TFGNN Schema, the name of the output `.tfrecords` file, Infer analysis results in `report.json`, and slicing criteria in CSV. The script is called as follows:

   ```
   find /tmp/infer-out/raw-ecpg -mindepth 1 -type d | python3.8 \
   ../model/schemas/feature_engineering.py \
   ../model/schemas/mixed_nodes/extended_cpg.pbtxt \
   /tmp/infer-out/graphs.tfrecords /tmp/infer-out/report.json \
   /tmp/infer-out/slicing_info.csv
   ```

8. The final step is to call the `model_inference.py` script (described below) as follows:

   ```
   python3.8 model_inference.py \
   ../model/schemas/mixed_nodes/extended_cpg.pbtxt /tmp/infer-out/ \
   graphs.tfrecords \
   ../model/saved_models/8_AUC_0.818/combined_AUC_0.818 \
   /tmp/infer-out/report.json /tmp/infer-out/ranked_report.json
   ```

   The output of this script is the `/tmp/infer-out/ranked_report.json` file, which contains the sorted reports from `report.json` according to the score from the GNN model.

The `model_inference.py` script applies the GNN model to the graphs, saved in the `graphs.tfrecords` file, and ranks individual reports from `report.json` according to the obtained scores. The script is a modified version of the `evaluate_model.py` script (see Section 5.8). Its input consists of 5 positional arguments:

1. TFGNN schema,
2. graphs in `.tfrecords` format,
3. directory containing the GNN model,
4. Infer output in `report.json`,
5. result file name.

Since the current models have not yet achieved significant results in the area of cross-analysis, the inference pipeline remains unused for now. Therefore, it has not been tested on real projects. However, the only project-specific part is the compiler wrapper, which was thoroughly tested on a range of real software in the author's bachelor's thesis [3], and before its incorporation into the `csmock` tool [21, 20], the functionality of the wrapper was tested on 55 randomly selected SRPM packages in the C language. The runtime of the inference pipeline depends primarily on the time taken for Infer analysis and the individual parts of the pipeline, which were described in previous chapters.

Chapter 6

Experimental Evaluation

bud This chapter describes the experimental evaluation of the developed GNN models for ranking reports from Meta Infer based on the probability of being a true positive. Specifically, Section 6.1 provides a detailed description of the architecture and hyperparameters of the base model, from which other models are derived. Section 6.2 discusses and evaluates the modifications of the base model on validation data. Section 6.3 compares the best developed models with existing models on test data. Section 6.4 compares the best developed GNN model with the *large language model* ChatGPT. Section 6.5 evaluates the developed models on cross-analysis. Finally, Section 6.6 summarizes and discusses the achieved results, and also describes possible future improvements.

6.1 Base Model

The general architecture and its main components used in the following models were already described in Section 4.1.5. Therefore, only supplementary information will be provided here, describing the specific architecture and hyperparameters of the base model – the model from which all other models mentioned in Section 6.2 are derived. The descriptions of basic machine learning concepts throughout this chapter, such as *loss function*, *dropout*, *binary cross entropy*, etc., are taken from [14], where they are discussed in detail and are only briefly mentioned here, as they are used in their conventional forms.

The architecture of the base model is shown in Figure 6.1. The layer initializing hidden states is of type `Dense(16)` (i.e., a densely-connected neural network layer with 16 outputs) with an activation function `relu`, for each node set. This is followed by 8 GNN layers of the type `MtAlbis`. Initial parameters were chosen primarily based on [80] and examples in the TFGNN repository[1]. Parameters such as `units` and `message_dim` were selected considering the batch size and GPU memory size. Some initial parameters were chosen randomly and for parameters not mentioned here, default values were retained. All (except for the last) `MtAlbis` layers share the same parameters, which are:

- `units=16` – size of the hidden states.
- `message_dim=16` – size of the messages on the edges.

[1] **TFGNN**'s repository: https://github.com/tensorflow/gnn.

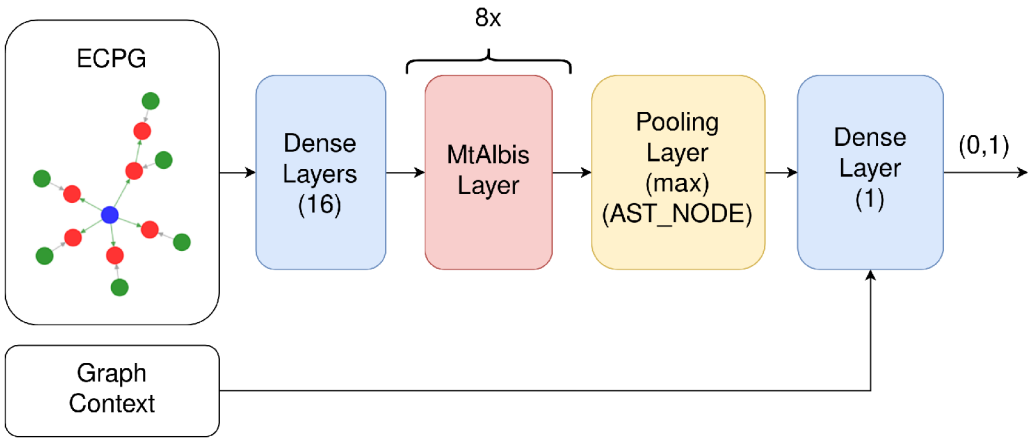

Figure 6.1: The figure shows the architecture of the base model (`Model 1`), which forms the basis for all other models developed in this thesis.

- `receiver_tag=tfgnn.TARGET` – specifies the direction of message passing, here it is along the direction of the edges (`tfgnn.SOURCE` would be in the opposite direction).
- `node_set_names=None` – updates nodes of all node set types. For the last layer, the value is set to `AST_NODE` in order to modify only nodes from the `AST_NODE` node set from which the following pooling layer reads. Hidden states of other node sets are discarded by the pooling layer, making it unnecessary to update their values in the last round of message passing.
- `state_dropout_rate=0.1` – the dropout rate applied to the pooled and combined messages from all edges.
- `simple_conv_reduce_type='mean|sum'` – the type of message aggregation.
- `next_state_type=residual` – can be set to `dense` or `residual`, where `residual` adds a residual connection from the old to the new node state.

Following the `MtAlbis` layers is a `Pool` layer of type `max`, which reads hidden states only from nodes of the `AST_NODE` node set. Here, for example, it could read from the root of the AST tree, where information would accumulate when changing the orientation of the `AST` edges (as mentioned in Section 4.1.4), but this would require the tree depth to be equal to or less than the number of GNN layers so that information from leaf nodes could reach the root. Since the depth varies and ECPGs are not just trees, it utilizes all `AST_NODE` nodes whose information is eventually aggregated using the `Pool` layer.

The context features, along with the output of the `Pool` layer, are inputs to the `Dense(1)` layer with the *sigmoid activation function*, which transforms the values into the range $(0, 1)$. The base model uses the `Adam` *optimizer* with a learning rate of `0.000002`. The loss function used is `BinaryCrossentropy`. The model is trained in batches of size `11` (limited due to GPU memory) over `300` epochs, where the number of steps per epoch is the dataset size divided by batch size. The model employs `EarlyStopping` with `patience` set at `20` (thus, the models are not trained for 300 epochs but only for tens of epochs, see Section 6.2). The base model is trained separately on each project from D2A – this represents a form of *3-fold validation*.

Although the architecture is trained for binary classification, the goal of the models is ranking, not classification. Therefore, it does not make sense to monitor separate metrics such as *precision*, *recall*, or *accuracy* in this case, since these are designed specifically for classification. These metrics are also not suitable for unbalanced data. The metric that appropriately reflects ranking and can be used for unbalanced data is the Area Under the Receiver Operating Characteristic Curve (AUROCC). The ROCC [64] plots the *True Positive Rate* (i.e., recall) on the Y-axis and the *False Positive Rate* on the X-axis for each classification threshold (previously mentioned metrics use only a single threshold), thereby clearly describing the ranking ability of the model. The True Positive Rate is defined as (here true positives and false positives relate to the model, not to the results of Infer):

$$True\ Positive\ Rate\ (TPR) = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Negatives\ (FN)}$$

The False Positive Rate is then defined as:

$$False\ Positive\ Rate\ (FPR) = \frac{False\ Positives\ (FP)}{False\ Positives\ (FP) + True\ Negatives\ (TN)}$$

AUROCC for a random model is 0.5 (indicated by a dashed line in all subsequent figures, see Figure 6.2) and for a perfect model is 1 (a model that can perfectly separate the classes). For the reasons mentioned above, AUROCC on validation data is monitored for early stopping in all trained models – the models are thus trained to achieve the highest possible AUROCC.

6.2 Hyperparameters Tuning

After the base model is created, it is necessary to tune its hyperparameters to adapt it to the specific task – in this case, the ranking of reports from Infer. If, even after tuning the hyperparameters, the architecture does not yield satisfactory results, a different architecture is typically tried. As will be evident from the results below, the architecture of the base model achieved very good results during the hyperparameter tuning process. Therefore, there was an effort to optimally tune this architecture. Due to limited computing resources, it was not possible to use automated hyperparameter tuning, which typically involves training many models with different settings. Thus, a manual approach had to be used, which requires fewer computing resources compared to automatic tuning but relies on experience and knowledge in the field.

A total of 14 models were trained, the results on validation data and their number of parameters are in Table 6.1. The models are trained and tested, due to limited computing resources, only on the smallest projects – httpd, libtiff, and nginx. The AUROCC of each model is either the average validation AUROCC across the individual projects or the validation AUROCC on a set of validation data composed of all the tested projects, as detailed below.

The following description includes a list of changes for each model compared to the previous model:

1. `Model 1` – base model, described in Section 6.1.

2. `Model 2` – increased network complexity – hidden state size increased to 18, an additional `MtAlbis` layer added, a `Dense(4)` added for context features, and a `Dense(8)` layer added before the final `Dense(1)` layer. The increase in complexity required reducing the batch size to 6. Also, the state dropout was increased to 0.15.

3. `Model 3` – reduced network complexity – decreased the size of hidden states to 12 but increased the batch size to 10.

4. `Model 4` – reducing complexity led to much worse results, so complexity was further increased – hidden state size increased to 20 at the cost of removing one `MtAlbis` layer and reducing the batch size to 6. A `Dropout(0.15)` layer was also added right after the `Pool` layer. The learning rate was decreased to 0.000001.

5. `Model 5` – added one `MtAlbis` layer (total of 9) at the cost of reducing the size of hidden states to 18.

6. `Model 6` – instead of training 3 models for each project, `Model 6` (and all subsequent models) is trained on all 3 projects at once (combining their training and validation sets). The learning rate was substantially increased to `0.0001`, and the `Dropout` layer was removed from the GNN head.

7. `Model 7` – increased state dropout to 0.2, tried only `mean` for `simple_conv_reduce`, and switched to `dense` for `next_state_type`.

8. `Model 8` – since `Model 7` experienced a significant drop in AUROCC, `Model 6` was restored. Only the state dropout was kept at 0.2 and an L2 regularization was added with a value of `0.00001` since the training AUROCC for `Model 6` was nearly 0.95 – the model manages to learn on training data, now it needs to better generalize.

9. `Model 9` – set edge dropout (in `MtAlbis` layers) to 0.2.

10. `Model 10` – the edge dropout led to a significant deterioration, so it was set back to 0. However, the state dropout was increased to 0.25, and the learning rate was decreased to 0.00005.

11. `Model 11` – again tried the so-far best `Model 8` but in a bi-directional mode – the direction of message passing in the `MtAlbis` layers is alternated using the `receiver_tag` parameter (see Section 6.1).

12. `Model 12` – again tried `Model 8` and utilized edge features – `ARGUMENT_INDEX` (until now all edge features were ignored).

13. `Model 13` – again tried `Model 8` but with *attention* – trainable message aggregation in `MtAlbis` layers. Used type `gat_v2` with 3 attention heads (4 are default, but the size of the hidden state – 18 – must be divisible by the number of attention heads).

14. `Model 14` – slightly increased the state dropout to 0.22.

The models have only been briefly described. Their source files can be found in the GitHub repository in commits labeled as, for example, `Model 8 - AUC 0.818`[2] (the best-performing

[2]Source code of `Model 8`: https://github.com/TomasBeranek/but-masters-thesis/blob/fdcaa8e5f896d50c9b55a616cea84d56a058d45f/model/src/mixed_nodes_model.py.

Table 6.1: The table shows the results of hyperparameter tuning and the size of each model. Validation data from httpd, libtiff, and nginx projects were used for the evaluation.

Model	Parameters	AUROCC
Model 1	96,515	0.630
Model 2	137,499	0.668
Model 3	61,941	0.557
Model 4	150,093	0.607
Model 5	137,499	0.598
Model 6	137,499	0.787
Model 7	106,071	0.632
Model 8	137,499	0.818
Model 9	137,499	0.775
Model 10	137,499	0.793
Model 11	140,523	0.786
Model 12	140,451	0.788
Model 13	109,563	0.816
Model 14	109,563	0.746

model). However, these historical versions of the training script (described in Section 5.7) were in the development stage and should only serve as a reference for the definition of the model architectures.

From Table 6.1, it is evident that the best performing models are `Model 8`, `Model 13`, and `Model 10`, respectively. All these models are very small – with less than 140 thousand parameters (which is about ~500KB on disk) – yet they achieve very good results. These models were trained (hardware specifications used are in Chapter 5) for 69 epochs (~460s per epoch), 7 epochs (~1,350s per epoch), and 73 epochs (~450s per epoch), respectively.

6.3 Models Comparison

As previously mentioned, this thesis compares with the models developed in [94, 68] which also focus on reducing false positives reported by Infer. For this comparison, the three best-performing models on the validation data, specifically `Model 8`, `Model 10`, and `Model 13`, were selected based on Table 6.1. Additionally, a `3-soft-vote` model was created, which ranks based on a *soft score* – the sum of the scores from the three top-performing models. Moreover, a `6-soft-vote` model comprising the six top-performing models (`Model 6`, `8`, `10`, `11`, `12`, and `13`) was also created. It is important to note that the models are not compared on identical test sets as the Graph D2A contains fewer samples than the original D2A dataset due to:

1. Support for only certain error types (see Section 4.1.1). The number of unsupported samples is ~1.6 % of the total D2A samples, thus minimally influencing the results.

2. The inability to generate ECPG from some D2A samples. These cases are significantly more frequent and could more substantially impact the results. Their quantity varies depending on the project, ranging from 3 % to 23 % for tested projects (see Table 5.1).

Table 6.2: A comparison of the existing models `vote`, `c-bert`, and `vote-new` with the models developed in this thesis. The comparison criterion is AUROCC on test data.

Model	httpd	libtiff	nginx
vote	0.77	0.89	0.77
c-bert	0.82	0.94	0.89
vote-new	0.90	0.98	0.93
Model 8	0.80	0.95	0.94
Model 10	0.79	0.91	0.91
Model 13	0.74	0.87	0.83
3-soft-vote	0.80	0.96	0.95
6-soft-vote	0.83	0.96	0.94

Table 6.2 presents a comparison of the `Model 8`, `Model 10`, `Model 13`, `3-soft-vote`, and `6-soft-vote` developed in this thesis with the existing `vote`, `c-bert`, and `vote-new` models from [94, 68]. The models are compared based on AUROCC on the test data, which is the only common metric across all models. The comparison is only shown for the projects httpd, libtiff, and nginx due to limited computational resources. The `vote`, `c-bert`, and `vote-new` models are trained on training and validation data, whereas the models developed in this thesis are trained only on training data with validation data used for early stopping.

From Table 6.2, it is evident that the developed GNN models can match or even surpass the state-of-the-art models, especially for nginx. However, the results for httpd are lower, likely due to a lack of data. As indicated in Table 6.1, models using a combined training set (`Model 6` and above) achieve significantly better results. It is possible that compared to existing models, these GNNs require more training data. The httpd project has the fewest samples in the original D2A dataset, and an additional ~22 % of samples were removed when generating Graph D2A from httpd, which greatly complicates learning.

However, models can also be compared from other perspectives, such as their size, which relates to the inference speed. All existing solutions are closed source, making it impossible to determine their sizes. Similarly, it is not possible to verify their results, experiment with the models, or use them. Hence, the models developed in this thesis are a promising open source alternative.

Specific ROC curves for the developed models on the combined test sets can be seen in Figure 6.2. ROC curves for individual projects are provided in the appendices in Figure C.1 (httpd), Figure C.2 (libtiff), and Figure C.3 (nginx).

The intended use case for these models is to rank Infer reports by likelihood of being a real error. Developers would then typically check only the most promising reports – for example, the top 5 % (the same value was chosen in [68]). Consider now the best-performing model (on average), `6-soft-vote`, which is deployed on the test data of the libtiff project. The percentage of real errors (equivalent to *precision* $*$ 100) in the libtiff project test data in Graph D2A is ~4.7 % (see Table 5.1). This number remains unchanged (on average) if a random 5 % of samples are checked – equivalent to ranking by a random model. However, if the top 5 % of samples according to the `6-soft-vote` model are selected, the amount of true positives increases to ~57.1 %. In terms of the number of samples – in unsorted reports, there will be on average 2.3 real errors for every 49 checked reports. In the sorted

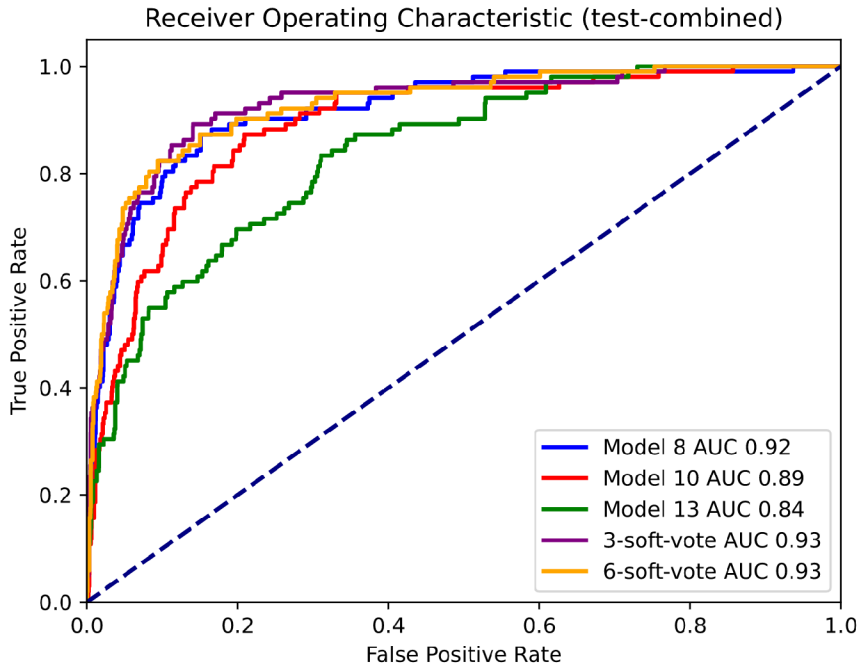

Figure 6.2: The figure shows the ROC curves for the top-performing models developed in this thesis, evaluated on a combined test set from the httpd, libtiff, and nginx projects.

reports, 28 real errors will be found for the same number of checked reports, which is more than 13 times as many.

Graphs showing precision values for different percentages of top samples and for top-performing models are presented in Figure C.4 (combined data from all projects), Figure C.5 (httpd), Figure C.6 (libtiff), and Figure C.7 (nginx). This metric becomes more sensitive as fewer top samples are considered.

6.4 Comparison with ChatGPT

In recent years, Large Language Models (LLMs) such as ChatGPT have become increasingly recognized by both professionals and the general public. ChatGPT can respond to textual inputs (and in version 4, even, e.g., image inputs) with textual outputs (again in version 4, even, e.g., image outputs). The introduction of ChatGPT [67] demonstrates the model's capabilities to search for and correct errors in code. The ability of ChatGPT (especially version 4) to handle programming tasks compared to other LLMs is discussed in [15], where ChatGPT4 is shown to be particularly effective. These results raise the question of how ChatGPT might perform in reducing false reports.

For the experiment, the `6-soft-model`, which on average achieved the best results in Section 6.3, was used. Ten true positives and ten false positives were randomly selected from the test set of the libtiff project. These samples, in their original JSON format (without class information), were submitted to a modified version of ChatGPT4 that could interpret code and search the internet, with the following instructions:

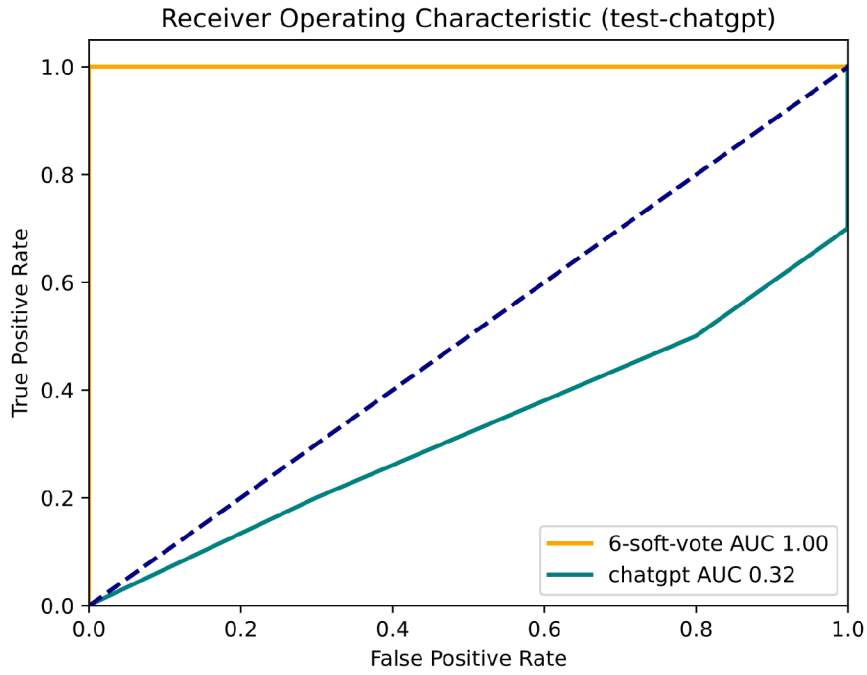

Figure 6.3: The figure shows ROC curves comparing `6-soft-vote` and ChatGPT4, on a randomly selected (with balanced classes) 20 samples from the test data of the libtiff project.

```
Behave like a binary classification model. You will receive a sample
from the D2A dataset, containing reports from Meta Infer static
analyzer. Your goal is to output a number in the range <0,1>. The
higher the number, the more certain you are that the report from Infer
is true. Individual samples contain the report itself, the location of
the error, codes of functions related to the error, and other useful
information.
```

From Figure 6.3, which displays the ROC curves for the scores from `6-soft-vote` and from ChatGPT version 4, it is apparent that ChatGPT4 exhibits random behavior in terms of ranking these selected samples. In contrast, `6-soft-vote` achieves a perfect score – distinguishing between the classes perfectly. When comparing the models in terms of their size, `6-soft-vote` again prevails with only 800 thousand parameters (comprising 6 sub-models), while ChatGPT4 has approximately 1.76 trillion parameters [57]. However, it is important to note that ChatGPT is a general-purpose model and not a classifier specifically designed for ranking reports from static analysis.

6.5 Cross-analysis

The ultimate goal of all models in the field of static analysis report filtering/ranking, based on the likelihood of being true positive, is to function on cross-analysis. Existing models `vote`, `c-bert`, and `vote-new` from [94, 68] are designed only for self-analysis – the model is

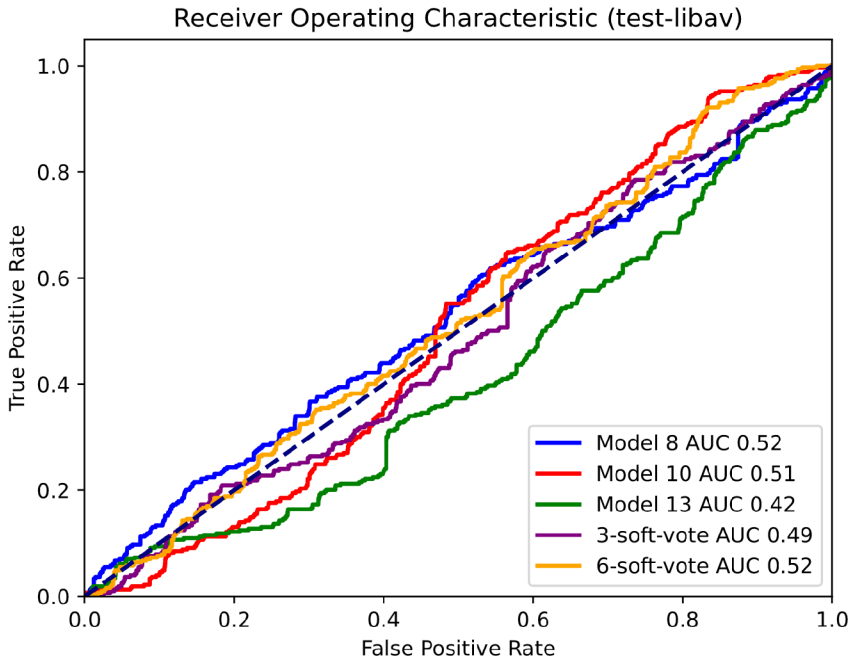

Figure 6.4: The figure shows ROC curves for the top performing models developed in this thesis in cross-analysis mode. The models were trained on the httpd, libtiff, and nginx projects and tested on the test data of the libav project.

trained and tested on the same project. However, the self-analysis is also useful in practice, especially for large projects with extensive git histories that can be used to train models.

A primary objective of feature engineering in Section 4.1.4 was to eliminate information that could lead the models to overfit to a specific project. To test cross-analysis capabilities, the top performing models were tested on the test data of the libav project. The results in Figure 6.4 indicate that the models, on average, exhibit random behavior. The developed models, like all existing ones, thus fail to function on cross-analysis, which represents a significant challenge in this research field.

6.6 Summary and Future Work

From the experiments in this chapter, it is evident that GNNs, and specifically the models developed in this thesis, are suitable for ranking reports from the Meta Infer static analyzer. The created models were able to match best existing solutions in this area that we are aware of, which were developed by strong industrial team from IBM. In the case of the nginx project, the existing models were even surpassed. However, results on the httpd project were weaker, which may be due to a lack of data for the httpd project, which is not only the smallest project in terms of the number of samples in the D2A dataset but also experienced high sample losses during the generation of Graph D2A. Nonetheless, we believe that these results demonstrate that the developed models are a promising open-source alternative to existing solutions, which are unfortunately all closed source.

The best-performing model, `6-soft-vote`, was also compared with the LLM model ChatGPT version 4. The model developed in this thesis proved superior with a perfect score, in contrast to ChatGPT4, which was unable to differentiate between false and real reports.

The models were also tested on a cross-analysis. None of the models were able to correctly distinguish false reports from real ones. Cross-analysis thus emerges as an unexpectedly challenging problem in this research area as no existing model that we are aware of functions effectively on cross-analysis either.

Future work should focus on testing self-analysis on all projects in the D2A dataset, which, however, requires training on a large number of samples and thus needs significant computational resources. Considering the results of experiments on the httpd project, it would be necessary to focus on improving the training pipeline to avoid such high sample losses. Specifically, it would be beneficial to increase the timeout for the LLVM Slicer tool and to focus on improving the success rate of LLVM bitcode extraction. Particularly, it should focus on the minority class – true positives (not just for httpd), which are naturally very scarce.

Various ways of future improvements have already been mentioned in previous chapters. Considering the results of the models on cross-analysis, it would be necessary to focus especially on adjusting feature engineering – more information that allows models to overfit on individual projects should be discarded. Additionally, refining the extraction of slicing criteria could help future graphs to contain less *noise* (i.e., redundant information).

There is also a plan to deploy the developed models as part of the csmock tool [21, 20], which allows the automatic running of various analyzers on SRPM packages. A plugin for the csmock tool that adds support for static analysis by Meta Infer was created in the author's bachelor's thesis [3]. It would simply involve supplementing this plugin with the developed models, which would rank the results of Infer.

Chapter 7

Conclusion

This thesis aimed to develop a machine learning-based system for ranking reports from the Meta Infer static analyzer based on their likelihood of being real error. Graph Neural Networks (GNNs) were selected due to their suitability for modeling various source code properties. The D2A dataset from IBM, which contains labeled Infer reports, was used for training. This dataset required conversion from a textual to a graphical format. To achieve this, a training pipeline was developed to produce Graph D2A – a graphical representation of D2A. This pipeline improves existing graph generation techniques by considering conditional compilation. The raw format of graphs in Graph D2A necessitated the design of a feature engineering process that optimizes and transforms these graphs into Extended Code Property Graphs (ECPGs), which enrich commonly used Code Property Graphs by including Call Graphs, data types, and other information.

Experimental results with GNN models trained on projects httpd (AUROCC 0.83), libtiff (AUROCC 0.96), and nginx (AUROCC 0.94) show that the developed models are competitive with existing state-of-the-art solutions created by strong industrial teams. The models even reached state-of-the-art results on the nginx project although they performed less well on the httpd project, likely due to a low number of samples. Nonetheless, these experiments show that the developed models are a promising open-source alternative since all existing solutions are closed-source. The models were also tested using cross-analysis, which unfortunately did not yield useful results. Cross-analysis remains a significant challenge as none of the existing models compared in this thesis function effectively in this mode either.

In this thesis, an inference pipeline was also developed for the automatic Infer analysis, construction of ECPGs, and model inference on real-world C (and subset of C++) software. Even if cross-analysis does not work, the inference pipeline could be utilized in the future for inference on the projects on which the models were trained.

Future work should focus on evaluating and fine-tuning the developed models on larger projects from the D2A dataset. Based on the experiment results from the httpd project, the training pipeline should be improved to minimize the loss of samples during transformation. Specifically, increasing the timeout for the LLVM Slicer tool and focusing on generating LLVM bitcode, especially for the minority class – real errors. There are also plans to deploy the developed models in the csmock tool, which automates analyses on SRPM packages.

Preliminary results of this thesis were presented at the Excel@FIT'24 conference, where it received an award from the expert panel.

Bibliography

[1] AlmaBetter. *Data Normalization in Machine Learning* online. 2023. 2023-06-08. Available at: https://www.almabetter.com/bytes/tutorials/data-science/normalization-in-machine-learning.

[2] August, D. *Lecture 2: Basic Control Flow Analysis* online. Princeton University, 2004-02-10. Available at: https://www.cs.princeton.edu/courses/archive/spr04/cos598C/lectures/02-ControlFlow.pdf. Subject: COS 598C Advanced Compilers.

[3] Beránek, T. *Practical Application of Facebook Infer on Systems Code*. Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/24187/.

[4] Berdine, J.; Calcagno, C. and O'Hearn, P. W. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: Boer, F.; Bonsangue, M.; Graf, S. and Roever, W., ed. *Proceedings of the 4th International Symposium on Formal Methods for components and Objects (FMCO)*. Amsterdam, Netherlands: Springer, Berlin, Germany, November 2005, p. 115–137. ISBN 978-3-540-36750-5.

[5] Beránek, T. *Graph D2A: D2A Dataset for Vulnerability Detection Transformed into ECPG Format for Training GNNs* online. Zenodo, 15. may 2024. Available at: https://doi.org/10.5281/zenodo.11187083.

[6] Brownlee, J. *SMOTE for Imbalanced Classification with Python* online. 17. march 2021. 2021-03-17. Available at: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

[7] Brownlee, J. *A Gentle Introduction to k-fold Cross-Validation* online. 04. october 2023. 2023-10-04. Available at: https://machinelearningmastery.com/k-fold-cross-validation/.

[8] Buratti, L.; Pujar, S.; Bornea, M.; McCarley, S.; Zheng, Y. et al. Exploring software naturalness through neural language models. *ArXiv preprint arXiv:2006.12641*, 2020, abs/2006.12641. Available at: https://arxiv.org/abs/2006.12641.

[9] Cao, S.; Sun, X.; Bo, L.; Wei, Y. and Li, B. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology*. Elsevier, 2021, vol. 136, p. 106576. ISSN 0950-5849. Available at: https://www.sciencedirect.com/science/article/pii/S0950584921000586.

[10] Chalupa, M. DG: A program analysis library. *Software Impacts*. Elsevier, 2020, vol. 6, p. 100038. ISSN 2665-9638. Available at: https://www.sciencedirect.com/science/article/pii/S2665963820300294.

[11] Chalupa, M. DG: analysis and slicing of LLVM bitcode. In: Hung, D. V. and Sokolsky, O., ed. *Automated Technology for Verification and Analysis: 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19–23, 2020, Proceedings 18*. Cham: Springer International Publishing, 2020, p. 557–563. ISBN 978-3-030-59152-6.

[12] Chalupa, M. *Llvm-slicer* online. Masaryk University, may 2021, 2021-05-06. Available at: https://github.com/mchalupa/dg/blob/master/doc/llvm-slicer.md. [cit. 2024-05-01].

[13] Cheng, X.; Wang, H.; Hua, J.; Xu, G. and Sui, Y. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM New York, NY, USA, april 2021, vol. 30, no. 3, p. 1–33. ISSN 1049-331X. Available at: https://doi.org/10.1145/3436877.

[14] Chollet, F. *Deep learning v jazyku Python*. Praha: Grada Publishing, a.s., 2019. Myslíme v ... ISBN ISBN 978-80-247-3100-1.

[15] Coello, C. E. A.; Alimam, M. N. and Kouatly, R. Effectiveness of ChatGPT in Coding: A Comparative Analysis of Popular Large Language Models. *Digital*. MDPI, 2024, vol. 4, no. 1, p. 114–125.

[16] Denisov, A. and Yamaguchi, F. *LLVM meets Code Property Graphs* online. LLVM Project, 23. february 2021. Available at: https://blog.llvm.org/posts/2021-02-23-llvm-meets-code-property-graphs/. [cit. 2024-04-11].

[17] Devlin, J.; Chang, M.-W.; Lee, K. and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *ArXiv preprint arXiv:1810.04805*, 2018. Available at: https://arxiv.org/abs/1810.04805.

[18] Distefano, D.; Fähndrich, M.; Logozzo, F. and O'Hearn, P. W. Scaling static analyses at Facebook. *Communications of the ACM*, 2019, vol. 62, no. 8, p. 62–70. ISSN 1557-7317.

[19] Duan, X.; Wu, J.; Ji, S.; Rui, Z.; Luo, T. et al. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 2019, p. 4665–4671. IJCAI'19. ISBN 9780999241141.

[20] Dudka, K. Fully Automated Static Analysis of Fedora Packages. In: *Flock*. Prague, Czech Republic: [b.n.], August 2014. Available at: https://kdudka.fedorapeople.org/static-analysis-flock2014.pdf.

[21] Dudka, K. Static Analysis and Formal Verification at Red Hat. In: *13th Alpine Verification Meeting (AVM'19)*. Brno, Czech Republic: [b.n.], September 2019. Available at: https://kdudka.fedorapeople.org/avm19.pdf.

[22] Ebalard, A.; Mouy, P. and Benadjila, R. Journey to a RTE-free X.509 parser. In: *Proceedings of the Symposium sur la sécurité des technologies de l'information et des communications(SSTIC)*. Rennes, France: [b.n.], June 2019, p. 171–200. ISBN 78-2-9551333-4-7.

[23] Emanuelsson, P. and Nilsson, U. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*. Elsevier, 2008, vol. 217, p. 5–21. ISSN 1571-0661. Available at: https://www.sciencedirect.com/science/article/pii/S1571066108003824. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008).

[24] Facebook, Inc.. *Analyzing apps or projects* online. Facebook, Inc., february 2015. Available at: https://fbinfer.com/docs/analyzing-apps-or-projects. [cit. 2023-01-29]. Infer v1.1.0.

[25] Facebook, Inc.. *Infer Static Analyzer* online. Facebook, Inc., february 2015. Available at: https://fbinfer.com/. [cit. 2023-01-29].

[26] Facebook, Inc.. *List of all issue types* online. Facebook, Inc., february 2015. Available at: https://fbinfer.com/docs/all-issue-types/. [cit. 2023-01-29]. Infer v1.1.0.

[27] Facebook, Inc.. *Separation logic and bi-abduction* online. Meta, february 2015. Available at: https://fbinfer.com/docs/separation-logic-and-bi-abduction/. [cit. 2023-01-29]. Infer v1.1.0.

[28] Fang, Y.; Huang, C.; Zeng, M.; Zhao, Z. and Huang, C. JStrong: Malicious JavaScript detection based on code semantic representation and graph neural network. *Computers & Security*. Elsevier, 2022, vol. 118, p. 102715. ISSN 0167-4048. Available at: https://www.sciencedirect.com/science/article/pii/S0167404822001110.

[29] Ferludin, O.; Eigenwillig, A.; Blais, M.; Zelle, D.; Pfeifer, J. et al. Tf-gnn: Graph neural networks in tensorflow. *ArXiv preprint arXiv:2207.03522*, 2022.

[30] Ganz, T.; Härterich, M.; Warnecke, A. and Rieck, K. Explaining Graph Neural Networks for Vulnerability Discovery. In:. New York, NY, USA: Association for Computing Machinery, 2021, p. 145–156. AISec '21. ISBN 9781450386579. Available at: https://doi.org/10.1145/3474369.3486866.

[31] Google LLC. *Protocol Buffers Documentation* online. 2021. 2021-12-21. Available at: https://protobuf.dev/. [cit. 2023-05-08].

[32] Graphviz. *DOT Language* online. Graphviz, 2022, 2022-10-04. Available at: https://graphviz.org/doc/info/lang.html. [cit. 2023-01-29].

[33] Guan, Z.; Wang, X.; Xin, W. and Wang, J. Code property graph-based vulnerability dataset generation for source code detection. In: Xu, G.; Liang, K. and Su, C., ed. *Frontiers in Cyber Security: Third International Conference, FCS 2020, Tianjin, China, November 15–17, 2020, Proceedings*. Singapore: Springer Singapore, 2020, p. 584–591. ISBN 978-981-15-9739-8.

[34] Hanif, H. and Maffeis, S. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022, p. 1–8.

[35] Hanif, H.; Nasir, M. H. N. M.; Ab Razak, M. F.; Firdaus, A. and Anuar, N. B. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications*. Elsevier, 2021, vol. 179, p. 103009. ISSN 1084-8045. Available at: https://www.sciencedirect.com/science/article/pii/S1084804521000369.

[36] Harmim, D. *Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer*. Brno, CZ, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/24185/.

[37] Herout, P. *Testování pro programátory*. 2nd ed. Kopp, 2016. ISBN 978-80-7232-481-1.

[38] Hvilshøj, F. *Introduction to Balanced and Imbalanced Datasets in Machine Learning* online. 11. november 2022. 2023-11-11. Available at: https://encord.com/blog/an-introduction-to-balanced-and-imbalanced-datasets-in-machine-learning/.

[39] IBM. *Sample Description and Dataset Stats* online. IBM, april 2021, 2021-04-14. Available at: https://github.com/IBM/D2A/blob/main/docs/dataset_stats.md. [cit. 2024-05-01].

[40] IBM Research. *D2A - Differential Analysis Dataset* online. IBM, 11. february 2021. Available at: https://developer.ibm.com/exchanges/data/all/d2a/. [cit. 2023-01-29].

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

[42] Jackson, W. *Static vs. dynamic code analysis: advantages and disadvantages* online. Government Media Executive Group LLC, 09. february 2009. Available at: https://gcn.com/cybersecurity/2009/02/static-vs-dynamic-code-analysis-advantages-and-disadvantages/287891/. [cit. 2023-01-29].

[43] Javatpoint. *Normalization in Machine Learning* online. 2022. 2022-01-27. Available at: https://www.javatpoint.com/normalization-in-machine-learning.

[44] Johnson, B.; Song, Y.; Murphy Hill, E. and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, New York, NY, USA, May 2013, p. 672–681. ISSN 1558-1225.

[45] Kitware, Inc.. *CMAKE_EXPORT_COMPILE_COMMANDS* online. December 2018, 2023-01-19. Available at: https://cmake.org/cmake/help/latest/variable/CMAKE_EXPORT_COMPILE_COMMANDS.html. [cit. 2023-01-29]. CMake v3.25.2.

[46] Kwangkeun, Y. *Inferbo: Infer-based buffer overrun analyzer* online. Meta, 06. february 2017. Available at: https://research.fb.com/blog/2017/02/inferbo-infer-based-buffer-overrun-analyzer/. [cit. 2023-01-29].

[47] Li, X.; Wang, L.; Xin, Y.; Yang, Y. and Chen, Y. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*. MDPI, 2020, vol. 10, no. 5, p. 1692. ISSN 2076-3417. Available at: https://www.mdpi.com/2076-3417/10/5/1692.

[48] Li, Y.; Tarlow, D.; Brockschmidt, M. and Zemel, R. Gated graph sequence neural networks. *ArXiv preprint arXiv:1511.05493*, 2015.

[49] Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y. et al. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*. IEEE, 2021, vol. 19, no. 4, p. 2821–2837.

[50] Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y. et al. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*. IEEE, 2021, vol. 19, no. 4, p. 2244–2258.

[51] Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H. et al. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, 2018, abs/1801.01681. Available at: http://arxiv.org/abs/1801.01681.

[52] Lin, G.; Zhang, J.; Luo, W.; Pan, L. and Xiang, Y. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2017, p. 2539–2541. CCS '17. ISBN 9781450349468. Available at: https://doi.org/10.1145/3133956.3138840.

[53] Liu, S.; Chen, Y.; Xie, X.; Siow, J. and Liu, Y. Retrieval-augmented generation for code summarization via hybrid gnn. *ArXiv preprint arXiv:2006.05405*, june 2020, p. arXiv:2006.05405.

[54] LLVM Project. *LLVM Bitcode File Format* online. LLVM Project, 2003, 2023-01-28. Available at: https://llvm.org/docs/BitCodeFormat.html. [cit. 2023-01-29].

[55] LLVM Project. *LLVM Language Reference Manual* online. LLVM Project, 2003, 2023-01-28. Available at: https://llvm.org/docs/LangRef.html. [cit. 2023-01-29].

[56] LLVM Project. *Llvm-link - LLVM bitcode linker* online. LLVM Project, 2003, 2024-05-03. Available at: https://llvm.org/docs/CommandGuide/llvm-link.html. [cit. 2024-05-03]. Llvm-link v17.0.0.

[57] Lubbad, M. *GPT-4 Parameters: Unlimited guide NLP's Game-Changer* online. Medium, 19. march 2023. Available at: https://medium.com/@mlubbad/the-ultimate-guide-to-gpt-4-parameters-everything-you-need-to-know-about-nlps-game-changer-109b8767855a. [cit. 2024-05-12].

[58] Marcin, V. *Statická analýza v nástroji Facebook Infer zaměřená na detekci uváznutí.* Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21920/.

[59] Marek, D. *Static Analysis Using Facebook Infer Focused on Errors in RCU-Based Synchronisation.* Brno, CZ, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/25138/.

[60] Menzli, A. *Graph Neural Network and Some of GNN Applications: Everything You Need to Know* online. Neptune Labs, 27. january 2023. Available at: https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications. [cit. 2023-01-29].

[61] Mikolov, T.; Chen, K.; Corrado, G. and Dean, J. Efficient estimation of word representations in vector space. *ArXiv preprint arXiv:1301.3781*, 2013.

[62] Møller, A. and Schwartzbach, M. I. *Static Program Analysis.* Department of Computer Science, Aarhus University, october 2018. Available at: http://cs.au.dk/~{}amoeller/spa/.

[63] Muske, T. B.; Baid, A. and Sanas, T. Review efforts reduction by partitioning of static analysis warnings. In: *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM).* Eindhoven, Netherlands: IEEE, New York, NY, USA, September 2013, p. 106–115.

[64] Narkhede, S. *Understanding AUC - ROC Curve* online. 26. june 2018. 2018-06-26. Available at: https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5.

[65] Neo4j, Inc.. *Native Graph Database | Neo4j Graph Database Platform* online. Neo4j, Inc., 08. may 2009. Available at: https://neo4j.com/product/neo4j-graph-database/. [cit. 2023-01-29].

[66] Oliveira, D. How to properly generate TFRecord files from your datasets. *Writing TFRecord Files the Right Way* online. 31. march 2021. 2021-01-31. Available at: https://pub.towardsai.net/writing-tfrecord-files-the-right-way-7c3cee3d7b12.

[67] OpenAI. *Introducing ChatGPT* online. OpenAI, 30. november 2022. Available at: https://openai.com/index/chatgpt/. [cit. 2024-05-12].

[68] Pujar, S.; Zheng, Y.; Buratti, L.; Lewis, B.; Chen, Y. et al. Analyzing source code vulnerabilities in the D2A dataset with ML ensembles and C-BERT. *Empirical Software Engineering.* Springer, 2024, vol. 29, no. 2, p. 48. Available at: https://link.springer.com/article/10.1007/s10664-023-10405-9.

[69] Rabheru, R.; Hanif, H. and Maffeis, S. A hybrid graph neural network approach for detecting PHP vulnerabilities. In: IEEE. *2022 IEEE Conference on Dependable and Secure Computing (DSC).* 2022, p. 1–9.

[70] Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J. et al. Automated vulnerability detection in source code using deep representation learning. In: IEEE. *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. 2018, p. 757–762.

[71] Saccente, N.; Dehlinger, J.; Deng, L.; Chakraborty, S. and Xiong, Y. Project achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network. In: IEEE. *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 2019, p. 114–121.

[72] Saxena, S. *Binary Cross Entropy/Log Loss for Binary Classification* online. Analytics Vidhya, 13. september 2023. 2023-09-13. Available at: https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/. [cit. 2024-05-09].

[73] Šikić, L.; Kurdija, A. S.; Vladimir, K. and Šilić, M. Graph Neural Network for Source Code Defect Prediction. *IEEE Access*. IEEE, 2022, vol. 10, p. 10402–10415.

[74] Soni, B. *Topic:11 Feature Construction & Splitting* online. 05. march 2023. 2023-03-05. Available at: https://medium.com/@brijesh_soni/topic-11-feature-construction-splitting-b116c60c4b2f#79b0. [cit. 2023-05-08].

[75] Suneja, S.; Zheng, Y.; Zhuang, Y.; Laredo, J. and Morari, A. Learning to map source code to software vulnerability using code-as-a-graph. *CoRR*, 2020, abs/2006.08614. Available at: https://arxiv.org/abs/2006.08614.

[76] TensorFlow. *Data Preparation and Sampling* online. TensorFlow, december 2023, 2023-12-14. Available at: https://github.com/tensorflow/gnn/commits/main/tensorflow_gnn/docs/guide/data_prep.md. [cit. 2024-05-06].

[77] TensorFlow. *Describing your Graph* online. TensorFlow, november 2023, 2023-11-16. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/docs/guide/schema.md. [cit. 2024-05-08].

[78] TensorFlow. *Input pipeline* online. TensorFlow, december 2023, 2023-12-11. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/docs/guide/input_pipeline.md. [cit. 2024-05-08].

[79] TensorFlow. *Introduction to GraphTensor* online. TensorFlow, december 2023, 2023-12-23. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/docs/guide/graph_tensor.md. [cit. 2024-05-06].

[80] TensorFlow. *Model Template „Albis“* online. TensorFlow, december 2023, 2023-12-13. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/models/mt_albis/README.md. [cit. 2024-05-09].

[81] TensorFlow. *Ragged tensors* online. TensorFlow, june 2023, 2023-06-07. Available at: https://www.tensorflow.org/guide/ragged_tensor. [cit. 2024-05-08].

[82] TensorFlow. *TF-GNN Modeling Guide* online. TensorFlow, july 2023, 2023-7-14. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/docs/guide/gnn_modeling.md. [cit. 2024-05-08].

[83] TensorFlow. *The TF-GNN Runner* online. TensorFlow, july 2023, 2023-07-14. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/docs/guide/runner.md. [cit. 2024-05-09].

[84] TensorFlow. *TFRecord and tf.train.Example* online. TensorFlow, september 2023, 2023-09-28. Available at: https://www.tensorflow.org/tutorials/load_data/tfrecord. [cit. 2024-05-05].

[85] TensorFlow. *TF-GNN Models* online. TensorFlow, march 2024, 2024-03-20. Available at: https://github.com/tensorflow/gnn/blob/main/tensorflow_gnn/models/README.md. [cit. 2024-05-09].

[86] The Clang Team. *Clang command line argument reference* online. 2007. Available at: https://clang.llvm.org/docs/ClangCommandLineReference.html. [cit. 2023-01-29]. Clang v17.0.0.

[87] The Joern Project. *Joern - The Bug Hunter's Workbench* online. The Joern Project, 17. april 2019. Available at: https://joern.io/. [cit. 2023-01-29].

[88] The Joern Project. *Overview | Joern Documentation* online. April 2019. Available at: https://docs.joern.io/. [cit. 2023-01-29].

[89] Tonder, R. and Goues, C. L. Static automated program repair for heap properties. In: *ICSE '18: Proceedings of the 40th International Conference on Software Engineering*. Gothenburg, Sweden: Association for Computing Machinery, New York, NY, USA, May 2018, p. 151–162. ISBN 978-1-4503-5638-1.

[90] Wu, Y.; Lu, J.; Zhang, Y. and Jin, S. Vulnerability detection in c/c++ source code with graph representation learning. In: IEEE. *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. 2021, p. 1519–1524.

[91] Xiaomeng, W.; Tao, Z.; Runpu, W.; Wei, X. and Changyu, H. CPGVA: code property graph based vulnerability analysis by deep learning. In: IEEE. *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*. 2018, p. 184–188.

[92] Yamaguchi, F.; Golde, N.; Arp, D. and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In: IEEE. *2014 IEEE Symposium on Security and Privacy*. 2014, p. 590–604.

[93] Yamaguchi, F.; Lottmann, M.; Schmidt, N.; Pollmeier, M.; Sharma, S. et al. *Code Property Graph Specification 1.1* online. 1.1th ed. The Joern Project, august 2023, 2023-08-15. Available at: https://cpg.joern.io/. [cit. 2024-05-04].

[94] Zheng, Y.; Pujar, S.; Lewis, B.; Buratti, L.; Epstein, E. et al. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In: IEEE. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2021, p. 111–120.

[95] Zhou, Y.; Liu, S.; Siow, J.; Du, X. and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*. Curran Associates, Inc., 2019, vol. 32. Available at: https://proceedings.neurips.cc/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf.

[96] Zou, D.; Wang, S.; Xu, S.; Li, Z. and Jin, H. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing*. IEEE, 2019, vol. 18, no. 5, p. 2224–2236.

[97] Eucalyp. *Automatic Icons*. Flaticon. Available at: https://www.flaticon.com/free-icons/automatic.

[98] wishforge.games. *Compile Compiler Script Code*. SVG Repo LLC. Available at: https://www.svgrepo.com/svg/384696/compile-compiler-script-code-config.

[99] Eucalyp. *Funnel*. Flaticon. Available at: https://www.flaticon.com/free-icon/information_249182.

[100] Freepik. *IBM icon*. Freepik. Available at: https://www.freepik.com/icon/ibm_5969147.

[101] Grafixpoint. *Knowledge Graph Icons*. Flaticon. Available at: https://www.flaticon.com/free-icons/knowledge-graph.

[102] Chang, T. *LLVM Icon*. LLVM. Available at: https://llvm.org/Logo.html.

[103] Long, J. *Logos*. Git. Available at: https://git-scm.com/downloads/logos.

[104] Freepik. *Neural Network Icons*. Flaticon. Available at: https://www.flaticon.com/free-icons/neural-network.

[105] Uniconlabs. *Preprocessing Icons*. Flaticon. Available at: https://www.flaticon.com/free-icons/preprocessing.

[106] Good Ware. *Risk*. Flaticon. Available at: https://www.flaticon.com/free-icon/risk_5999218.

[107] ShiftLeft Inc.. *ShiftLeft Inc.* Available at: https://github.com/ShiftLeftSecurity.

Appendix A

Contents of the Attached Memory Media

This appendix describes the contents of the attached memory media. The memory media contains:

- `d2a/` – the original D2A dataset, which can also be downloaded from [40].
- `d2a-bitcode/` – zipped LLVM bitcode for the httpd project – only for testing the generation of Graph D2A, so that LLVM bitcode does not have to be re-generated (it is computationally expensive).
- `d2a-filtered/` – filtered D2A dataset.
- `graph-d2a/` – zipped Graph D2A (raw ECPGs) for all projects from which TFRecords were generated, namely httpd, libtiff, nginx, and libav (the latter is divided into several parts as detailed below). So far, it has not been possible to find a place for online storage of the entire D2A Graph, which occupies hundreds of GBs when compressed. Once the D2A Graph is uploaded, a link will be provided in the `README.md` of the repository.
- `httpd-dependencies/` – already configured libraries necessary for generating LLVM bitcode for httpd.
- `repository/` – all the source files, which are also available on GitHub[1]
  - `dataset/` – source files for the D2A to Graph D2A transformation.
    - `construction_phase_d2a` – for generating Graph D2A, see Section 5.4.
    - `filter.py` – for filtering D2A, see Section 5.1.
    - `generate_bitcode.py` – for generating LLVM bitcode, see Section 5.2
    - `Makefile` – for automating the execution of these scripts.
  - `dev-utils/` – a set of scripts that do not belong to the main implementation but were used, for example, for data exploration and other useful tasks.
    - `concat_tfrecords.py` – for concatenating TFRecords files.

[1] GitHub repository: https://github.com/TomasBeranek/but-masters-thesis.

    - `extract_sample.py` – for extracting and displaying samples from D2A.
    - `find_unique_values.py` – for finding unique attribute values in Graph D2A.
    - `graphs_comparison.sh` – for comparing different versions of Graph D2A.
    - `records_counter.py` – for counting samples in TFRecords files.
    - `remove_duplicates.py` – for removing duplicate samples in D2A at the LLVM bitcode level.
    - `remove_invalid_symlinks.py` – for removing incorrect symlinks in LLVM bitcode (this issue has been fixed).
    - `stats.py` – for generating statistics about D2A – Tables C.1 and C.2.
- `experiments/` – a set of experiments that support and demonstrate various claims made in this thesis. If an experiment is executable, it can be run using the `make` command in its directory. Details of the experiment are provided for each experiment separately (typically in `Makefile`).
    - `arg-passing/` – demonstration of argument passing in raw ECPG.
    - `comparison-with-chatgpt/` – data for comparison with ChatGPT, from Section 6.4.
    - `compilation-from-D2A/` – demonstration that code cannot be compiled directly from D2A.
    - `entry-function/` – demonstration that Infer always reports the entry function in the `procedure` attribute.
    - `global-vars/` – demonstration of storing access to global variables in raw ECPG.
    - `include-headers-to-bitcode/` – demonstration that slicing criteria can also be specified for `.h` files because they are included in LLVM bitcode.
    - `joern-batch-processing/` – demonstration that Joern batch processing is equivalent to single sample processing.
    - `line-slicing/` – demonstration that LLVM slicer retains everything on the line specified as the slicing criterion.
    - `removing-duplicates/` – test to demonstrate the functionality of removing duplicate samples.
    - `speed-test/` – comparison of repository search types by speed when generating LLVM bitcode.
    - `struct-alias/` – demonstration of storing structures and aliases in raw ECPG.
- `inference-pipeline/` – source code of the inference pipeline.
    - `example/` – an artificial example to demonstrate the functionality of the inference pipeline.
    - `inference_pipeline` – inference pipeline, see Section 5.10.
    - `Makefile` – for installing/uninstalling compiler wrappers and running the experiment.
    - `model_inference.py` – for inference using GNN models, see Section 5.10.
    - `slicing_criteria_extraction.py` – for extraction of slicing criteria, from Section 5.3.

        * `wrapper` – template for compiler wrapper, see Section 5.9.
    - `model/` – source files for feature engineering, training, and testing models.
        * `schemas/` – source files for feature engineering.
            · `extended_cpg.pbtxt` – TFGNN schema for ECPG.
            · `feature_engineering.py` – feature engineering, see Section 5.6.
            · `find_normalization_coefficients.py` – for extraction of normalization coefficients, see Section 5.5.
            · `Makefile` – targets for automating the execution of these scripts.
        * `src/` – source files for models.
            · `evaluate_model.py` – for model evaluation, see Section 5.8.
            · `mixed_nodes_model.py` – for model training, see Section 5.7.
        * `Makefile` – targets for training models, evaluating models, and scenarios.
    - `sep-presentation-en/` – LaTeX source code of the SEP presentation.
    - `sep-text-en/` – LaTeX source code of the SEP text.
    - `text-en/` – LaTeX source code of this thesis.
    - `text-template/` – LaTeX source code of the used template.
    - `xberan46-2024.pdf` – PDF version of this thesis.
    - `.gitignore`
    - `README.md`
- `results/` – output graphs for all scenarios from Section 5.8.
- `saved_models/` – all trained models – `Model 1` to `Model 14`.
- `text-en/` – LaTeX source code of this thesis.
- `tfrecords/` – dataset in TFRecords format for all projects that were used for training or testing, namely httpd, libtiff, nginx, and libav. The dataset in this format was also published on Zenodo [5].
- `xberan46-2024.pdf` – PDF version of this thesis.

Appendix B

Installation and User Manual

This appendix contains the installation and user manual, which were tested on a clean, normal (i.e., not minimal) installation of Ubuntu 20.04.2.0 LTS. For training or evaluating GNN models, it is advisable to have a GPU compatible with the TensorFlow library, otherwise the computation will be significantly slower.

Installation

All source code for training and inference pipelines is in script form, so there is no need for any installation, except for installing a compiler wrapper using a make target (see below). However, it is necessary to install required dependencies.

LLVM Slicer can be installed as follows[1]:

```
sudo apt install git cmake make llvm zlib1g-dev clang g++ python3.8
git clone https://github.com/mchalupa/dg
cd dg
mkdir build && cd build
cmake ..
make -j4
sudo ln -s ${PWD}/tools/llvm-slicer /usr/bin/llvm-slicer
```

Joern can be installed using:

```
sudo apt install curl default-jdk default-jre
git clone https://github.com/joernio/joern
cd joern
sudo ./joern-install.sh
```

For LLVM2CPG, it is necessary to download the binary release for Ubuntu 20.04[2] and then install it using:

```
unzip llvm2cpg-0.8.0-LLVM-11.0-ubuntu-20.04.zip
```

[1] Official installation guide for LLVM Slicer: https://github.com/mchalupa/dg/blob/master/doc/compiling.md.

[2] LLVM2CPG's binary release for Ubuntu 20.04: https://github.com/ShiftLeftSecurity/llvm2cpg/releases/download/0.8.0/llvm2cpg-0.8.0-LLVM-11.0-ubuntu-20.04.zip.

```
mv llvm2cpg-0.8.0-LLVM-11.0-ubuntu-20.04/ llvm2cpg
sudo ln -s ${PWD}/llvm2cpg/llvm2cpg /usr/bin/llvm2cpg
```

To generate LLVM bitcode, the following dependencies need to be installed for various projects:

- httpd

  ```
  sudo apt install libpcre3 libpcre3-dev autoconf libtool-bin
  ```

- libtiff

  ```
  sudo apt install libgl-dev freeglut3-dev
  ```

- ffmpeg (the same applies to libav as they share some libraries)

  ```
  sudo apt install nasm yasm libsdl2-dev
  ```

- openssl

  ```
  sudo apt install perlbrew
  perlbew init # follow the instructions to finish the installation
  perlbrew install perl-5.28.0
  ```

It is also necessary to install Infer using the binary release[3]:

```
tar xf infer-linux64-v1.1.0.tar.xz
sudo ln -s ${PWD}/infer-linux64-v1.1.0/bin/infer /usr/bin/infer
```

Additionally, the following Python3.8 packages must be installed:

```
sudo apt install python3-pip
python3.8 -m pip install tqdm "pandas==1.3.4" "networkx==3.1" \
"matplotlib==3.4.3" "scikit-learn==1.2.0" "tensorflow-gnn==0.6.1"
```

The command line tool `parallel` also have to be installed:

```
sudo apt install parallel
```

For experiments, the following dependencies are necessary:

```
sudo apt install graphviz tree
```

User Manual

The training pipeline is divided into individual phases and is generated in parts due to its high computational demands. To simplify generation, a set of Makefiles has been created. However, using these Makefiles requires storing the individual outputs in directories exactly as defined below (or it is possible to modify the paths in the Makefiles, or call the tools

[3]Infer's binary release: https://github.com/facebook/infer/releases/download/v1.1.0/infer-linux64-v1.1.0.tar.xz.

without using Makefiles). The manual assumes that the starting working directory contains directories and files from the attached memory media.

The first step is to prepare D2A:

```
mv d2a repository/dataset/
```

D2A is then filtered:

```
cd repository/dataset/ && make filter-d2a
```

Next, the slicing criteria are extracted:

```
make slicing-info
```

Then, prepare a directory for the repositories of individual projects from D2A, add pre-configured libraries for httpd (it is also possible to download from official sites and configure it as described in Section 5.2), and download the original project repositories:

```
mkdir projects && mv ../../httpd-dependencies projects/
make download-repos
```

From this point forward, commands will only be listed for the httpd project; other projects are generated similarly unless otherwise noted. LLVM bitcode is generated using:

```
make bitcode-httpd-1
make bitcode-httpd-0
```

Before generating LLVM bitcode for openssl, it is necessary to switch the Perl version as follows:

```
perlbrew switch perl-5.28.0
```

After generating the LLVM bitcode, it is possible to start generating the D2A Graph:

```
mkdir -p graph-d2a/httpd_1 && make graph-httpd-1
mkdir -p graph-d2a/httpd_0 && make graph-httpd-0
```

The D2A Graphs for projects from which TFRecords were generated, namely httpd, libtiff, nginx, and libav, are zipped on the attached memory media in the `graph-d2a/` directory to avoid re-generation. Because the D2A Graph is not only computationally expensive to create but also memory-expensive to store, libav_0 is split into 3 `.zip` files, which must be transformed into TFRecords separately, and the results combined using the script `repository/dev-utils/concat_tfrecords.py` (an example of its usage is provided inside).

Normalization coefficients could be extracted here, but they have already been generated and are inserted directly in `feature_engineering.py`. However, for demonstration, it can be done using:

```
cd ../model/schemas && make extract-norm-coeffs-httpd
```

Extraction of normalization coefficients is also possible for libtiff and nginx. For other projects, make targets were not created as they were not used for training.

To generate TFRecords, enter (current working directory is `repository/model/schemas`):

```
mkdir ../tfrecords
make transform-httpd-1
make transform-httpd-0
```

Once TFRecords for the httpd, libtiff, and nginx projects are created, training can commence. As their generation is also time-consuming, they are included on the memory media in the `tfrecords` directory, or are available on Zenodo [5]. To start training the currently configured model (the architecture of `Model 8` – the best standalone model) with combined data, along with adding `tfrecord` from the media and also adding models from the media (`saved_models/`), use:

```
cd ..
mv ../../tfrecords .
mv ../../saved_models .
make train-combined-model
```

For example, `Model 8` can be evaluated on the test data using:

```
make evaluate-model-test ID=8
```

Alternatively, it is possible to run, for example, a `combined` scenario:

```
make scenario-combined
```

For a demonstration of the inference pipeline, a simple project was created in which Infer finds 3 errors – 1x `DEAD_STORE` (which is not supported as it is always true positive) and 2x `NULL_DEREFERENCE`. When executing the experiment, compiler wrappers are first installed, the project is compiled, the inference pipeline is run, and finally, the wrappers are uninstalled. The experiment can be initiated (assuming the working directory is `repository/model/`):

```
cd ../inference-pipeline
make
```

The output is `ranked_report.json`, where errors are scored (attribute `model_score`) and ranked using `Model 8`.

Appendix C

Additional Data

This appendix contains supplementary tables and figures that provide additional data relevant to the discussions and experiments presented in earlier chapters of the thesis.

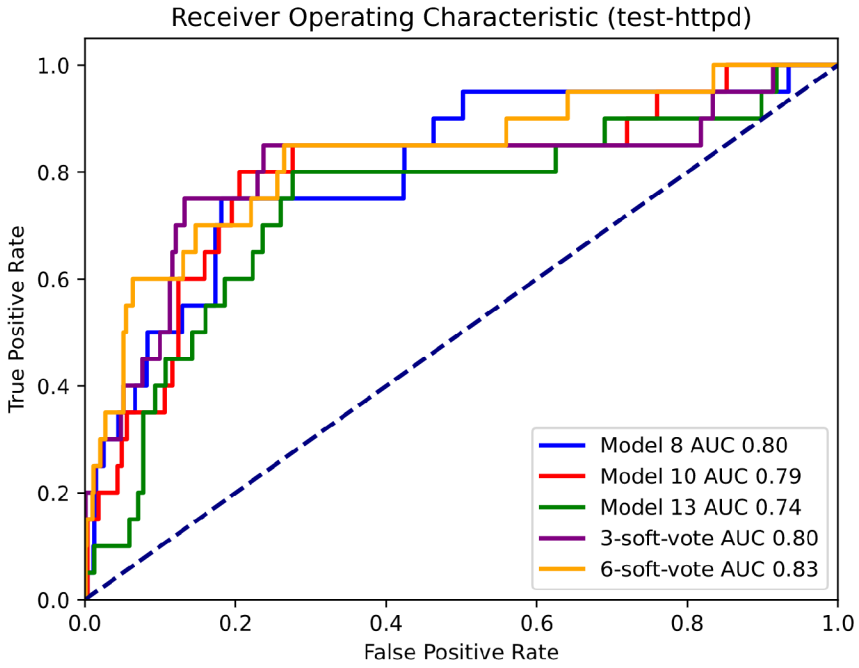

Figure C.1: The figure shows ROC curves for the top-performing models developed in this thesis. The models were evaluated on test data from the httpd project.

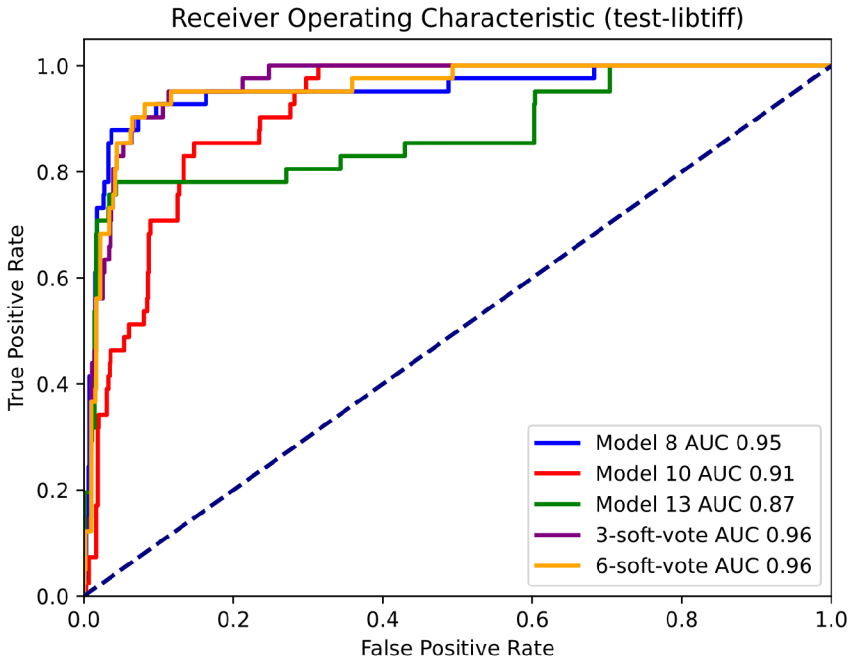

Figure C.2: The figure shows ROC curves for the top-performing models developed in this thesis. The models were evaluated on test data from the libtiff project.

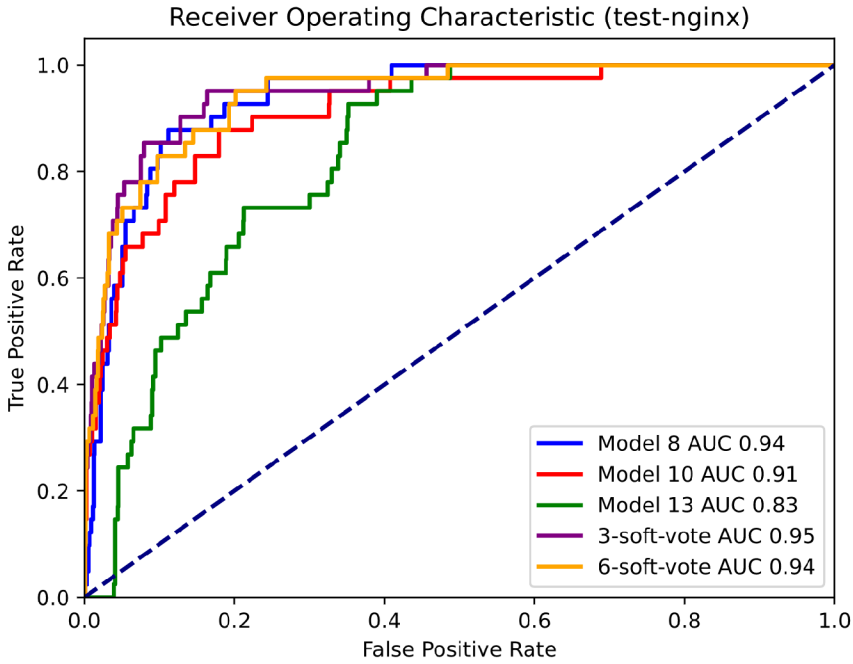

Figure C.3: The figure shows ROC curves for the top-performing models developed in this thesis. The models were evaluated on test data from the nginx project.

Table C.1: The table contains the distribution of all D2A samples (except the after fix type) by error type and label (true positive/false positive). This table shows the distribution for the openssl, libav, and nginx projects. Table C.2 shows the distribution for the remaining projects. The green highlighted rows represent the types of errors supported by the system designed in this thesis.

Error Type	**openssl**	**libav**	**nginx**
INTEGER_OVERFLOW_L5	4046/166221	2892/156942	162/4333
BUFFER_OVERRUN_L5	1656/81700	684/27403	39/2070
BUFFER_OVERRUN_L4	758/24928	165/9560	83/5016
INTEGER_OVERFLOW_U5	191/11015	178/9944	30/1254
NULLPTR_DEREFERENCE	125/10055	78/8580	2/132
BUFFER_OVERRUN_U5	297/14353	82/3944	67/2992
INTEGER_OVERFLOW_L2	178/8008	55/2819	20/625
NULL_DEREFERENCE	98/4336	24/4810	3/37
INFERBO_ALLOC_MAY_BE_BIG	49/734	237/3098	0/0
BUFFER_OVERRUN_L3	382/9391	16/805	6/366
UNINITIALIZED_VALUE	28/695	55/3526	3/116
BUFFER_OVERRUN_L2	80/594	97/1664	1/243
PULSE_MEMORY_LEAK	18/4046	0/0	1/586
DEAD_STORE	50/1355	15/1011	0/12
MEMORY_LEAK	18/3021	4/214	0/66
DANGLING_POINTER_DEREFERENCE	13/1353	2/636	0/3
BUFFER_OVERRUN_L1	14/319	9/767	0/3
DIVIDE_BY_ZERO	0/33	1/318	0/0
INTEGER_OVERFLOW_L1	11/235	3/200	2/22
USE_AFTER_FREE	2/391	15/47	1/9
INTEGER_OVERFLOW_R2	0/0	1/91	0/0
BUFFER_OVERRUN_S2	7/217	0/21	1/28
RESOURCE_LEAK	1/118	1/11	0/28
PREMATURE_NIL_TERMINATION_ARGUMENT	0/0	0/0	0/0
INFERBO_ALLOC_IS_ZERO	0/0	0/4	0/1
INFERBO_ALLOC_IS_BIG	0/3	0/0	0/1
DEALLOCATE_STACK_VARIABLE	0/14	0/0	0/0
INFERBO_ALLOC_MAY_BE_NEGATIVE	0/1	0/0	0/2
BIABD_USE_AFTER_FREE	0/11	0/0	0/0
BUFFER_OVERRUN_R2	0/0	0/0	0/0
POINTER_TO_INTEGRAL_IMPLICIT_CAST	0/1	0/0	0/0
All Types	8022/343148	4614/236405	421/17945

Table C.2: The table contains the distribution of all D2A samples (except the after fix type) by error type and label (true positive/false positive). This table shows the distribution for the libtiff, httpd, and ffmpeg projects. Table C.1 shows the distribution for the remaining projects. The green highlighted rows represent the types of errors supported by the system designed in this thesis.

Error Type	**libtiff**	**httpd**	**ffmpeg**
INTEGER_OVERFLOW_L5	306/5917	64/2632	2912/394952
BUFFER_OVERRUN_L5	101/2468	45/1534	590/110655
BUFFER_OVERRUN_L4	9/682	28/835	190/25546
INTEGER_OVERFLOW_U5	67/562	21/1338	241/32493
NULLPTR_DEREFERENCE	2/70	14/2521	201/24812
BUFFER_OVERRUN_U5	27/524	25/2477	142/19479
INTEGER_OVERFLOW_L2	7/488	8/99	101/10391
NULL_DEREFERENCE	2/210	0/388	74/10228
INFERBO_ALLOC_MAY_BE_BIG	4/5	0/0	70/9113
BUFFER_OVERRUN_L3	9/334	0/48	47/1615
UNINITIALIZED_VALUE	0/111	3/59	151/5549
BUFFER_OVERRUN_L2	0/0	0/27	44/2572
PULSE_MEMORY_LEAK	6/141	0/11	0/0
DEAD_STORE	0/117	7/115	24/1582
MEMORY_LEAK	2/121	0/87	3/604
DANGLING_POINTER_DEREFERENCE	0/121	0/142	7/1742
BUFFER_OVERRUN_L1	0/5	1/13	4/1255
DIVIDE_BY_ZERO	4/166	0/0	11/776
INTEGER_OVERFLOW_L1	0/9	1/3	5/595
USE_AFTER_FREE	0/0	0/18	1/251
INTEGER_OVERFLOW_R2	0/0	0/0	3/561
BUFFER_OVERRUN_S2	7/32	0/0	3/73
RESOURCE_LEAK	0/0	0/12	1/26
PREMATURE_NIL_TERMINATION_ARGUMENT	0/0	0/116	0/0
INFERBO_ALLOC_IS_ZERO	0/7	0/0	1/3
INFERBO_ALLOC_IS_BIG	0/0	0/0	0/10
DEALLOCATE_STACK_VARIABLE	0/0	0/0	0/0
INFERBO_ALLOC_MAY_BE_NEGATIVE	0/6	0/0	0/2
BIABD_USE_AFTER_FREE	0/0	0/0	0/0
BUFFER_OVERRUN_R2	0/0	0/0	0/6
POINTER_TO_INTEGRAL_IMPLICIT_CAST	0/0	0/0	0/0
All Types	553/12096	217/12475	4826/654891

Table C.3: The table shows all selected node sets and their attributes during the feature selection phase. It also indicates which node sets will be merged. The symbol '✓' denotes an attribute that is used, the symbol '✗' denotes an unused attribute, '-' indicates that the attribute does not exist, and '0' indicates that the attribute does not exist but will be filled with zeros for the purpose of merging node sets.

Node Set	LABEL	ARGUMENT INDEX	CODE	ORDER	FULL NAME	IS EXTERNAL	New Node Set
METHOD	✓	0	✗	✓	✓	✓	AST_NODE METHOD_INFO
METHOD PARAMETER IN	✓	0	✗	✓	-	-	AST_NODE
METHOD RETURN	✓	0	✗	✓	-	-	AST_NODE
MEMBER	✓	-	✗	✓	-	-	MEMBER
TYPE	✓	-	-	-	✓	-	TYPE
BLOCK	✓	✓	✗	✓	-	-	AST_NODE
CALL	✓	✓	✗	✓	-	-	AST_NODE
FIELD IDENTIFIER	✓	✓	✗	✓	-	-	AST_NODE
IDENTIFIER	✓	✓	✗	✓	-	-	AST_NODE
LITERAL	✓	✓	✓	✓	-	-	AST_NODE LITERAL_INFO
LOCAL	✓	0	✗	✓	-	-	AST_NODE
METHOD REF	✓	✓	✗	✓	-	-	AST_NODE
RETURN	✓	✓	✗	✓	-	-	AST_NODE
UNKNOWN	✓	✓	✗	✓	-	-	AST_NODE

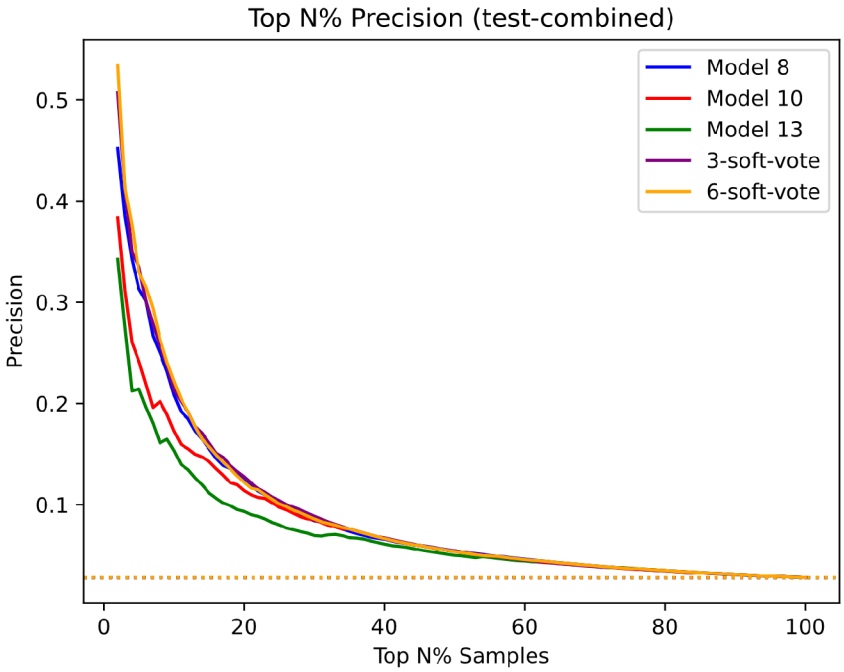

Figure C.4: The figure shows the precision of top-performing models for various percentages of top-ranked samples. The models were evaluated on combined test data from the httpd, libtiff, and nginx projects. The dashed horizontal line indicates the precision of a random model.

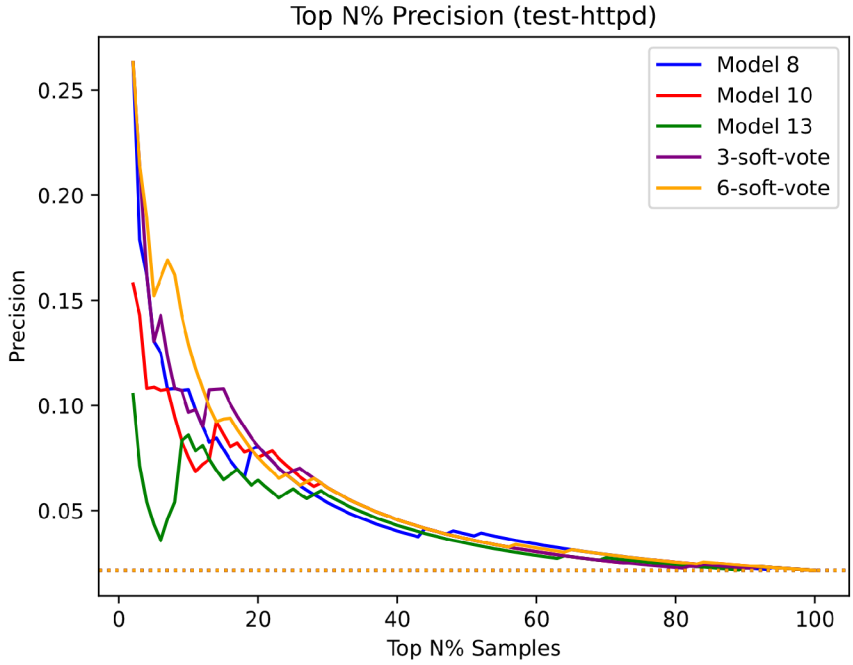

Figure C.5: The figure shows the precision of top-performing models for various percentages of top-ranked samples. The models were evaluated on test data from the httpd project. The dashed horizontal line indicates the precision of a random model.

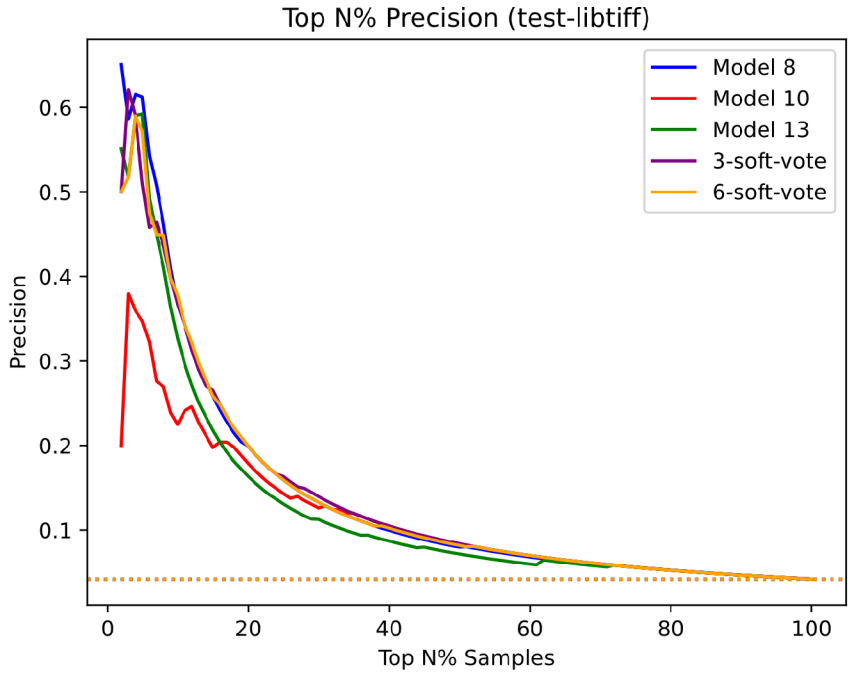

Figure C.6: The figure shows the precision of top-performing models for various percentages of top-ranked samples. The models were evaluated on test data from the libtiff project. The dashed horizontal line indicates the precision of a random model.

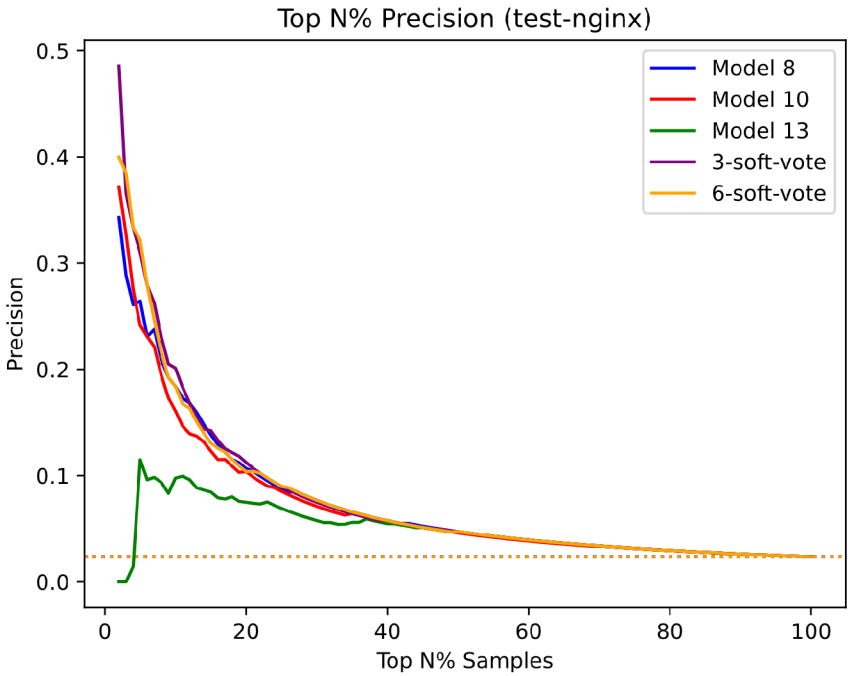

Figure C.7: The figure shows the precision of top-performing models for various percentages of top-ranked samples. The models were evaluated on test data from the nginx project. The dashed horizontal line indicates the precision of a random model.

Table C.4: The table shows all attributes for every node set present in Graph D2A. This table is divided into multiple parts by columns; this is Part 1. The remaining parts are in Tables C.5, C.6, C.7, C.8, and C.9.

Node Set	ID	LABEL	LINE NUMBER	CODE	COLUMN NUMBER	ORDER	NAME
META DATA	✗	✗	-	-	-	-	-
FILE	✗	✗	✗	✗	✗	✗	✗
NAMESPACE	✗	✗	✗	✗	✗	✗	✗
NAMESPACE BLOCK	✗	✗	✗	✗	✗	✗	✗
METHOD	✗	✓	✗	✗	✗	✓	✗
METHOD PARAMETER IN	✗	✓	✗	✗	✗	✓	✗
METHOD PARAMETER OUT	✗	✗	✗	✗	✗	✗	✗
METHOD RETURN	✗	✓	✗	✗	✗	✓	-
MEMBER	✗	✓	✗	✗	✗	✓	✗
TYPE	✗	✓	-	-	-	-	✗
TYPE DECL	✗	✗	✗	✗	✗	✗	✗
BLOCK	✗	✓	✗	✗	✗	✓	-
CALL	✗	✓	✗	✗	✗	✓	✗
FIELD IDENTIFIER	✗	✓	✗	✗	✗	✓	-
IDENTIFIER	✗	✓	✗	✗	✗	✓	✗
LITERAL	✗	✓	✗	✓	✗	✓	-
LOCAL	✗	✓	✗	✗	✗	✓	✗
METHOD REF	✗	✓	✗	✗	✗	✓	-
RETURN	✗	✓	✗	✗	✗	✓	-
UNKNOWN	✗	✓	✗	✗	✗	✓	-

Table C.5: The table shows all attributes for every node set present in Graph D2A. This table is divided into multiple parts by columns; this is Part 2. The remaining parts are in Tables C.4, C.6, C.7, C.8, and C.9.

Node Set	ARGUMENT INDEX	ARGUMENT NAME	TYPE FULL NAME	DYNAMIC TYPE HINT FULL NAME	FILENAME	FULL NAME
META DATA	-	-	-	-	-	-
FILE	-	-	-	-	-	-
NAMESPACE	-	-	-	-	-	-
NAMESPACE BLOCK	-	-	-	-	✗	✗
METHOD	-	-	-	-	✗	✓
METHOD PARAMETER IN	-	-	✗	✗	-	-
METHOD PARAMETER OUT	-	-	✗	-	-	-
METHOD RETURN	-	-	✗	✗	-	-
MEMBER	-	-	✗	✗	-	-
TYPE	-	-	-	-	-	✓
TYPE DECL	-	-	-	-	✗	✗
BLOCK	✓	✗	✗	✗	-	-
CALL	✓	✗	✗	✗	-	-
FIELD IDENTIFIER	✓	✗	-	-	-	-
IDENTIFIER	✓	✗	✗	✗	-	-
LITERAL	✓	✗	✗	✗	-	-
LOCAL	-	-	✗	✗	-	-
METHOD REF	✓	✗	✗	✗	-	-
RETURN	✓	✗	-	-	-	-
UNKNOWN	✓	✗	✗	✗	-	-

Table C.6: The table shows all attributes for every node set present in Graph D2A. This table is divided into multiple parts by columns; this is Part 3. The remaining parts are in Tables C.4, C.5, C.7, C.8, and C.9.

Node Set	SIGNATURE	METHOD FULL NAME	PARSER TYPE NAME	EVALUATION STRATEGY	HASH	AST PARENT FULL NAME
META DATA	-	-	-	-	✗	-
FILE	-	-	-	-	✗	-
NAMESPACE	-	-	-	-	-	-
NAMESPACE BLOCK	-	-	-	-	-	-
METHOD	✗	-	-	-	✗	✗
METHOD PARAMETER IN	-	-	-	✗	-	-
METHOD PARAMETER OUT	-	-	-	✗	-	-
METHOD RETURN	-	-	-	✗	-	-
MEMBER	-	-	-	-	-	-
TYPE	-	-	-	-	-	-
TYPE DECL	-	-	-	-	-	✗
BLOCK	-	-	-	-	-	-
CALL	✗	✗	-	-	-	-
FIELD IDENTIFIER	-	-	-	-	-	-
IDENTIFIER	-	-	-	-	-	-
LITERAL	-	-	-	-	-	-
LOCAL	-	-	-	-	-	-
METHOD REF	-	✗	-	-	-	-
RETURN	-	-	-	-	-	-
UNKNOWN	-	-	✗	-	-	-

Table C.7: The table shows all attributes for every node set present in Graph D2A. This table is divided into multiple parts by columns; this is Part 4. The remaining parts are in Tables C.4, C.5, C.6, C.8, and C.9.

Node Set	AST PARENT TYPE	IS EXTERNAL	INDEX	IS VARIADIC	COLUMN NUMBER END	LINE NUMBER END
META DATA	-	-	-	-	-	-
FILE	-	-	-	-	-	-
NAMESPACE	-	-	-	-	-	-
NAMESPACE BLOCK	-	-	-	-	-	-
METHOD	✗	✓	-	-	✗	✗
METHOD PARAMETER IN	-	-	✗	✗	-	-
METHOD PARAMETER OUT	-	-	✗	✗	-	-
METHOD RETURN	-	-	-	-	-	-
MEMBER	-	-	-	-	-	-
TYPE	-	-	-	-	-	-
TYPE DECL	✗	✗	-	-	-	-
BLOCK	-	-	-	-	-	-
CALL	-	-	-	-	-	-
FIELD IDENTIFIER	-	-	-	-	-	-
IDENTIFIER	-	-	-	-	-	-
LITERAL	-	-	-	-	-	-
LOCAL	-	-	-	-	-	-
METHOD REF	-	-	-	-	-	-
RETURN	-	-	-	-	-	-
UNKNOWN	-	-	-	-	-	-

Table C.8: The table shows all attributes for every node set present in Graph D2A. This table is divided into multiple parts by columns; this is Part 5. The remaining parts are in Tables C.4, C.5, C.6, C.7, and C.9.

Node Set	TYPE DECL FULL NAME	ALIAS TYPE FULL NAME	CONTAINED REF	CLOSURE BINDING ID	CANONICAL NAME	DISPATCH TYPE
META DATA	-	-	-	-	-	-
FILE	-	-	-	-	-	-
NAMESPACE	-	-	-	-	-	-
NAMESPACE BLOCK	-	-	-	-	-	-
METHOD	-	-	-	-	-	-
METHOD PARAMETER IN	-	-	-	-	-	-
METHOD PARAMETER OUT	-	-	-	-	-	-
METHOD RETURN	-	-	-	-	-	-
MEMBER	-	-	-	-	-	-
TYPE	✗	-	-	-	-	-
TYPE DECL	-	✗	-	-	-	-
BLOCK	-	-	-	-	-	-
CALL	-	-	-	-	-	✗
FIELD IDENTIFIER	-	-	-	-	✗	-
IDENTIFIER	-	-	-	-	-	-
LITERAL	-	-	-	-	-	-
LOCAL	-	-	-	✗	-	-
METHOD REF	-	-	-	-	-	-
RETURN	-	-	-	-	-	-
UNKNOWN	-	-	✗	-	-	-

Table C.9: The table shows all attributes for every node set present in Graph D2A. This table is divided into multiple parts by columns; this is Part 1. The remaining parts are in Tables C.4, C.5, C.6, C.7, and C.8.

Node Set	LANGUAGE	OVERLAYS	ROOT	VERSION	INHERITS FROM TYPE FULL NAME
META DATA	✗	✗	✗	✗	-
FILE	-	-	-	-	-
NAMESPACE	-	-	-	-	-
NAMESPACE BLOCK	-	-	-	-	-
METHOD	-	-	-	-	-
METHOD PARAMETER IN	-	-	-	-	-
METHOD PARAMETER OUT	-	-	-	-	-
METHOD RETURN	-	-	-	-	-
MEMBER	-	-	-	-	-
TYPE	-	-	-	-	-
TYPE DECL	-	-	-	-	✗
BLOCK	-	-	-	-	-
CALL	-	-	-	-	-
FIELD IDENTIFIER	-	-	-	-	-
IDENTIFIER	-	-	-	-	-
LITERAL	-	-	-	-	-
LOCAL	-	-	-	-	-
METHOD REF	-	-	-	-	-
RETURN	-	-	-	-	-
UNKNOWN	-	-	-	-	-