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

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR — **Bc. TOMÁŠ BERÁNEK**
AUTOR PRÁCE

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

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

148712

Institut:	Department of Intelligent Systems (UITS)
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:	2022/23

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. Propose and implement a machine learning-based approach (using the dataset obtained in step 3) that will be able to assess the likelihood that an issue reported by Infer represents a true positive.
5. Evaluate your solution on at least 2 different open-source projects.
6. 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.

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

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Consultants:	Mgr. Marek Grác, Ph.D. Ing. Viktor Malík
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	3.11.2022

Abstract

Static analysis is now commonly used in the development process to detect vulnerabilities. However, static analyzers are often faced with a high number of false positives. Therefore, a major effort is being made for automatic detection of false positives. This paper deals with the development of a detection system for Meta Infer tool, where the false positive rate reaches almost 90 %. The detection system is based on graph neural networks, which have proven to be a very powerful tool for finding vulnerabilities in source code. The tool is composed of two main parts – the construction of the graph representation and the detection model. This term project focuses mainly on the first part and describes the design of a system for automatic construction of code property graphs from C/C++ source files. The proposed graph construction system can connect to a running compilation process and automatically extract the LLVM internal representation, from which the code property graph is then constructed. This graph is further sliced according to the Infer reports to contain only the information useful for later report classification. By using the LLVM internal representation, the output graph is independent of the language and even the static analyzer.

Abstrakt

Statická analýza je dnes běžně využívána ve vývojovém procesu pro detekci zranitelností. Statické analyzátory se však často potýkají s vysokým počtem falešných hlášení. Proto se vyvíjí obrovská snaha pro automatickou detekci falešných hlášení. Tato práce se zabývá vytvořením detekčního systému pro nástroj Meta Infer, kde jejich podíl dosahuje téměř 90 %. Detekční systém je založen na grafových neuronových sítích, které se ukázaly být velmi silným nástrojem pro hledání zranitelností ve zdrojových kódech. Nástroj je složen ze dvou hlavních částí – konstrukce grafové reprezentace a detekčního modelu. Tento semestrální projekt se zaměřuje zejména na první část a popisuje návrh systému pro automatické konstruování grafů vlastností kódů z C/C++ zdrojových souborů. Navržený systém pro konstrukci grafů se dokáže napojit na běžící překladový proces a automaticky z něj extrahovat LLVM interní reprezentaci, ze které je poté vytvořen graf vlastností kódu. Tento graf je dále prořezán podle hlášení Inferu, aby obsahoval pouze informace užitečné pro pozdější klasifikaci hlášení. Výstupní graf je díky použití LLVM interní reprezentaci nezávislý na jazyce a dokonce i na statickém analyzátoru.

Keywords

static analysis, Meta Infer, deep learning, graph neural networks, false positive detection, code property graphs, csmock, Joern, LLVM Slicer, program slicing, LLVM, LLVM2CPG, SRPM package, graph extraction, vulnerability detection

Klíčová slova

statická analýza, Meta Infer, hluboké učení, grafové neuronové sítě, detekce falešných hlášení, grafy vlastností kódu, csmock, Joern, LLVM Slicer, prořezávání programů, LLVM, LLVM2CPG, SRPM balíček, extrakce grafů, detekce zranitelností

Reference

BERÁNEK, Tomáš. *Evaluating Reliability of Static Analysis Results Using Machine Learning.* Brno, 2023. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

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
February 6, 2023

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

1 Introduction **3**

2 Preliminaries **5**
2.1 Static Analysis 5
2.2 Meta Infer 6
2.3 Graph Neural Networks 8
2.4 Source Code as a Graph 9
2.5 LLVM-Slicer 12
2.6 LLVM2CPG 13
2.7 Joern 13

3 Constructing Graphs from Source Code **14**
3.1 Capture Phase 16
3.2 Graph Construction Phase 18

4 Future Work **21**
4.1 Automation of a Graph Construction 21
4.2 Construction of Graphs from D2A Dataset 21
4.3 Graph Neural Network Architecture 22
4.4 Self-training 22
4.5 Integration with Csmock 23
4.6 Experiments 24

5 Conclusion **25**

Bibliography **26**

List of Figures

2.1 Abstract syntax tree for the code in Listing 2.1. This figure was taken from [61]. 10
2.2 Control flow graph (on the left) and program dependence graph (on the right) for the code in Listing 2.1. These figures were taken from [61]. 11
2.3 Code property graph for the code in Listing 2.1. This figure was taken from [61]. 12

3.1 A simplified block diagram of a complete pipeline for constructing code property graphs from C source files. The dashed line indicates intermediate products. The diagram uses icon from [17]. 14
3.2 A block diagram of the capture phase of the pipeline for extracting code property graphs from C source files. The dashed line indicates intermediate products. The diagram uses icons from [17, 65]. 16
3.3 A block diagram of the graph construction phase of the pipeline for extracting code property graphs from C source files. The dashed line indicates intermediate products. The diagram uses icons from [56, 66, 67]. 18

4.1 An illustration of the basic usage of the csmock tool for automated analysis of SRPM packages. The figure was taken from [13]. 23

Chapter 1

Introduction

Static analysis is a widely used technique for finding *vulnerabilities* during software development. *Static analyzers* can also be deployed on code that is not yet finished. This makes it possible to detect vulnerabilities in the early stages of development, even before tests can be run. However, static analyzers often suffer from a high number of *false positives*. 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. Compared to other static analyzers, it is characterized by its ease of use – its input is compilation commands that compile the analyzed source files. Although it is a very useful tool, 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 almost 90 % of the reports are false positives.

The main contribution of the future Master's thesis is the design and implementation of a *false positive detection system* for the Meta Infer tool. If reports that are likely to be false positives could be detected and subsequently removed, it would be possible to reduce the amount of false positives that Infer produces. Thus, the proposed detection system could make Infer a more practical tool because the current percentage of false positives is too high. The input to the detection system is the source files and list of vulnerabilities from Infer. The output will then be, for each vulnerability found by Infer, the probability that it is a false positive. The detection system is based on *graph neural networks* (GNN), which have proven to be very powerful tool for finding vulnerabilities in source code. The detection system is composed of two parts – the graph construction and the GNN detection model.

The main contribution of this term project is the design of the first part of the false positive detection system – the construction of graphs from source code. This part is designed as a pipeline that connects to the running *compilation* process of the analyzed software and constructs graphs from the compiled source files. The output of this pipeline is *code property graphs* (CPG) – a very widely used graphical representation of code for vulnerability detection that captures the *syntactic* and *semantic* properties of the code. The pipeline internally uses the LLVM intermediate representation (LLVM IR), from which it then constructs, for each vulnerability found by Infer, a single CPG that represents the vulnerability.

The use of LLVM IR makes the output graph independent of the pipeline input language. And since the vulnerability found by Infer is only used in the pipeline to extract criteria for the *program slicing*, the resulting graph is also independent of the used static analyzer.

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

Structure of the term project. The rest of this term project is structured as follows. Chapter 2 explains basic principles of static analysis and graph neural networks. The chapter also describes code property graphs and tools used in this term project, namely Meta Infer, LLVM-Slicer, LLVM2CPG and Joern. Chapter 3 describes the design of a pipeline for automatic extraction of code property graphs from source files. Chapter 4 describes future work, specifically *automation* of the proposed pipeline, transformation of the D2A dataset to code property graphs, selection of a graph neural network architecture, *self-training* technique, integration with the csmock tool, and the data for the experiments. This term project together with the future work presented in Chapter 4 will form the Master's thesis.

Chapter 2

Preliminaries

This chapter introduces the basic concepts, principles and tools on which this term project builds. Specifically, Section 2.1 briefly describes static analysis itself, its applications, advantages and limitations. Section 2.2 describes the Meta Infer static analyzer, its use, types of *detectable vulnerabilities*, 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 a 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 [1, 15, 27] can be understood as a *reasoning* of *run-time properties* of computer programs without the need to run them or provide their inputs. Using static analysis, it is possible to investigate program properties such as *time* or *memory complexity*, look for *security risks* such as *null pointer dereference*, *access beyond array boundaries*, improper handling of resources, etc. It is also possible to check for *synchronization errors* such as *deadlock*, *data race*, *atomicity violation*, 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 inserting possible 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 [1, 29]:

- implicitly consider all possible paths in the code (even the ones with rare execution path),
- can report the exact location of the vulnerability and thus speed up the fix,

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

- do not require executable, sometimes even compilable source code, so bugs can be detected early in development,
- after the initial setup, they can be run fully automatically.

However, static analysis also has its disadvantages [27, 29]:

- static analyzers usually do not have information about *functional requirements* and thus cannot be used for *validation*,
- initial setup can be tedious for some tools, as it may require e.g. creating *models* of certain functions, accessing compilation of a code, or manually editing the required style guide,
- static analysis also cannot be used to check the *semantic function* of a program (e.g. does the program give the correct result?),
- running static analyzers can be very time and memory consuming,
- static analyzers can report false positives (false reports) or *false negatives* (missed real errors).

Rice's theorem implies [46] 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 *approximation*. 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 [14]. 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 [17] (formerly Facebook Infer) is an open-source[4] *framework* for writing *intraprocedural* and *interprocedural* static analyses [26, 43, 44]. Although it is a framework, Infer already includes a number of default and non-default (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 vulnerabilities, e.g. InferBO, which uses the *symbolic interval* technique [32] to detect incorrect array indexing, or the Bi-abduction plugin, which uses *bi-abduction* [19] – a form of *inference* for *separation logic* that models computer memory – to detect vulnerabilities associated with incorrect memory manipulation. Among other things, Infer can detect *null-pointer dereference*, *dead store*, *uninitialized value*, *deadlock*, *data race*, *variable overflow*, and many other types of vulnerabilities. Table 2.1 lists all the plugins that Infer provides, along with information about language support and whether the plugin is enabled by default. More detailed information about each plugin and the types of vulnerabilities reported by Infer can be found at [18].

[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	✓	✓	✓			✓

Infer is not a *sound*, which in the context of finding vulnerabilities means that it may miss some (it may have a false negatives). Instead, it aims for maximal practical use - scaling to millions of lines of code thanks to *modular analysis*. It is also very simple to use [16] 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 [2, 58]. 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 [1] show that this support is not complete 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 vulnerabilities. Experiments on *real-world* programs in previous work [1] also show that Infer has a very high number of false positives. Specific numbers suggest approximately 4.5 false positives for every real vulnerability. 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 C language while using conditional compilation. Without dead stores the number increases to approximately 9 false positives for every real vulnerability. The high number of false positives in static analyzers results

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

in developers' distrust of these tools and consequent ignoring of analysis results [9, 30, 47]. Therefore, efforts are made to reduce false positives.

2.3 Graph Neural Networks

There are a number of approaches for detecting vulnerabilities in programs using *machine learning* [25]. However, when focusing on the area of *deep learning*, approaches can be divided into *convolutional neural networks* (CNN) [11], *recurrent neural networks* (RNN) [35, 36, 37, 38, 52, 64] and graph neural networks (GNN) [4, 7, 21, 53, 54, 63], depending on the *architecture* of the model used. These approaches are often combined with each other [20, 33, 49, 51].

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 [45]. In order to use the source code as input to a CNN, it must first be transformed into a graph (e.g. AST) and then into a matrix (e.g. *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 a adjacency matrix (which has a fixed order of nodes) in multiple ways. This property is very undesirable because we want the same result 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.

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 [3, 24], 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 a picture or a sequence is supported by the experiments in [54], especially on *synthetic datasets* (on *datasets* with real-world examples, all approaches failed in a given experiment).

For this term project, the most important thing is to define the format of the input graph. To make it clear what the format is based on, a brief description of a general graph neural network follows. The following description uses a slightly modified notation from [34].

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 output node of edge $e = (v, v')$ is v and the input node is v'. The *embedding vector* of 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, we define auxiliary sets of nodes. The set $\textsc{In}(v) = \{v' | (v', v) \in \mathcal{E}\}$ contains the *predecessors* of node v. The set $\textsc{Out}(v) = \{v' | (v, v') \in \mathcal{E}\}$ contains the *descendants* of node v. *Propagation* then proceeds by updating each node until *convergence* using the following formula:

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

Where the function f can be a *linear function* $\mathbf{h}_{v'}^{(t)}$ or a *neural network*. The output of this network is for each node 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 *Almedia-Pineda algorithm* is used to learn such a model. Firstly, the algorithm runs the model until the values of $\mathbf{h}_v^{(T)}$ converge. These values must be *constrained* to guarantee convergence. Since the whole GNN is differentiable, it is then possible to compute *gradients* for parameter adjustments from the converged solution.

2.4 Source Code as a Graph

There are many types of graphs that are used as source code representations, e.g. *abstract syntax trees* (AST), *control flow graphs* (CFG), *program dependency graphs* (PDG) and others. A very common type is the code property graph (CPG), which is composed of all three previous graphs and used in its pure form in e.g. [39, 54]. Modified versions of it are often used as well, e.g. *simplified CPGs* (SCPG) for *function-level vulnerability detection* in C/C++ [59], CPGs with added edges that reflect the original order of *tokens* (individual source code elements) [63] or *code composite graphs* (CCG) again for vulnerability detection in C/C++ [4]. Furthermore, e.g. PDGs alone are used for finding *malicious code* in JavaScript [20], XFGs (*subgraph* of PDG) for detecting vulnerabilities in C/C++ code [7] or CFGs together with token sequences for detecting vulnerabilities in PHP [50].

According to [61], the reason for the creation of CPGs is the inability of each subgraph type to detect certain types of vulnerabilities independently during *traversal*. For example, AST is not suitable for detecting *division by zero*. However, by combining AST and PDG it is possible, but still cannot detect e.g. integer overflow. This can only be detected by combining AST, CFG and PDG. 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 vulnerabilities in it. Exceptions are e.g. *race conditions*, which need more external information. A complete table is given in [61]. The following graph definitions are based on the original definitions from the paper introducing CPGs [61], with only minor changes in notation (to resemble the GNN definition given earlier) and changes in some notations to make them formally correct.

To formally define a CPG, it is first necessary to define a *property graph* [61], which is a commonly used graph type in *graph databases* such as Neo4j. Formally, a property graph is an oriented *multigraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \lambda, \mu)$, here $\mathcal{V}$ denotes the set of nodes, $\mathcal{E}$ denotes the set of edges, λ denotes the *edge labeling function* $\lambda : \mathcal{E} \to \Sigma$, where $\Sigma = 1, ..., L_{\mathcal{E}}$ are the edge labels. Finally, μ denotes the function that assigns attributes to nodes and edges $\mu : (\mathcal{V} \cup \mathcal{E}) \times K \to S$, where K is the set of *attribute* names and S is the set of attribute values.

An AST [61] 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. An AST constructed for this code is shown in Figure 2.1. To create a formal CPG description,

```
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: Code sample. The code was taken from [61].

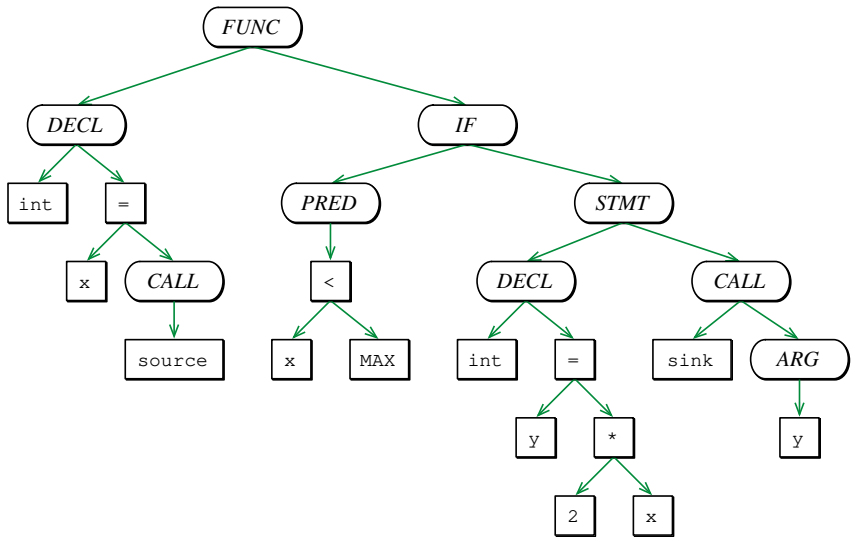

Figure 2.1: Abstract syntax tree for the code in Listing 2.1. This figure was taken from [61].

the subgraph types must be converted to the same format - the previously defined property graph. 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 $\mu_{\text{AST}} : \mathcal{V}_{\text{AST}} \times K_{\text{AST}} \rightarrow S_{\text{AST}}$ 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 AST e.g. *variable*, *constant*, mathematical operators etc. And S_{order} which assigns values (order) to nodes in the AST to preserve the ordering from the original tree.

A CFG [61] 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. A 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:

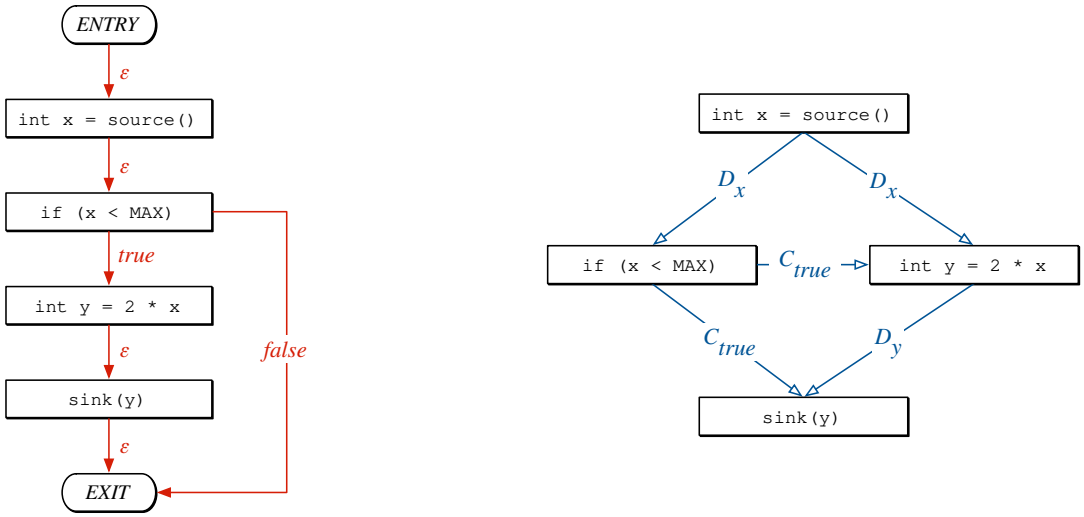

Figure 2.2: Control flow graph (on the left) and program dependence graph (on the right) for the code in Listing 2.1. These figures were taken from [61].

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

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

A PDG [61] is again an oriented graph whose nodes are statements and predicates. There are two types of edges in PDG, namely *data dependency edges*, which model the influence of a variable on the value of another variable, and *control dependency edges*, which 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 all nodes are 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 '*data*' to the '*symbol*' edges as the name of the variable represented by the default 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:

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

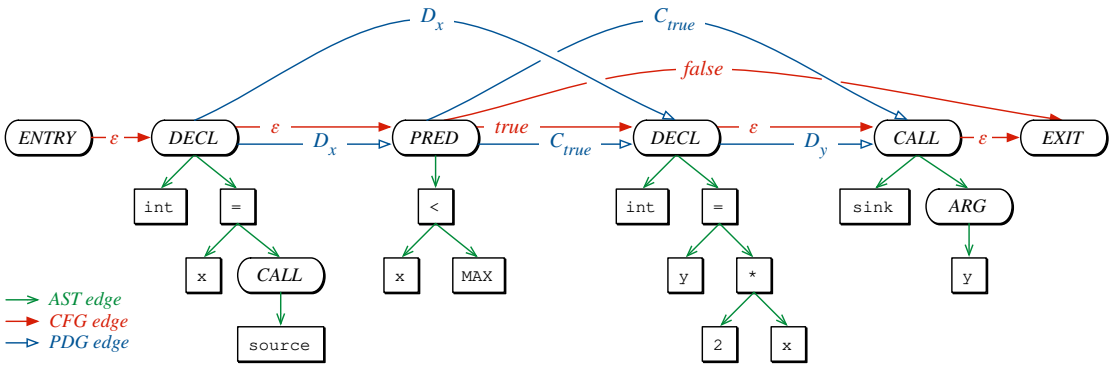

Figure 2.3: Code property graph for the code in Listing 2.1. This figure was taken from [61].

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 μ function is:

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

A CPG for the code in Listing 2.1 is shown in Figure 2.3.

2.5 LLVM-Slicer

LLVM-slicer is an open-source[6] tool using DG library [5, 6]. The DG library implements various interprocedural static analyses namely *pointer analysis*, d*ata 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 only LLVM bitcode [40]. LLVM bitcode is a *storage format* for LLVM IR[7] [41], 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 DG analysis for program-slicing - removing pieces of code that have no effect on *user-defined* areas in the code. Experiments and results of slicing success on benchmarks from the Software Verification Competition can be found in [6]. 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

[6] **LLVM-Slicer**' repository: https://github.com/mchalupa/dg.
[7] **LLVM intermediate representation (LLVM IR)**.

able to slice it. The input to the LLVM-slicer is a single LLVM bitcode file and the output is the sliced LLVM bitcode.

2.6 LLVM2CPG

LLVM2CPG is an open-source[8] tool for constructing CPGs (defined in Section 2.4) from LLVM bitcode. The original CPG was created for high-level languages such as C, which creates some problems when creating CPGs from low-level LLVM IR [8]. 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. the original 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[9] (proprietary), Plume[10] (open-source) or Joern (open-source, see Section 2.7).

2.7 Joern

Joern [56] is a very powerful open-source[11] platform providing various tools in the area of static analysis. Using Joern, it is possible to write custom static analyses or *queries* over source files. Joern supports the *programming languages* C, C++, JavaScript, Kotlin, Python and Java. It is also possible to analyze Java bytecode or binary programs for the x86 architecture. 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 [22] or **csv for the Neo4j graph database** [48]. 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*. Although Joern is a very diverse tool, it is only used in this term project as a format *transformer* and is therefore described here very briefly.

[8] **LLVM2CPG**'s repository: https://github.com/ShiftLeftSecurity/llvm2cpg.
[9] **Ocular**'s documentation: https://docs.shiftleft.io/ocular/quickstart.
[10] **Plume**'s documentation: https://plume-oss.github.io/plume-docs/.
[11] **Joern**'s repository: https://github.com/joernio/joern.

Chapter 3

Constructing Graphs from Source Code

This chapter describes the design of a pipeline for constructing code property graphs from real-world C programs. Specifically, the introduction of this chapter describes the disadvantages of current approaches, the motivation for creating this pipeline, and its general principle. Section 3.1 describes the principle of connecting the pipeline to the build process and preprocessing the inputs of the graph construction phase. Section 3.2 describes the principle of constructing graphs from LLVM bitcode and Meta Infer outputs.

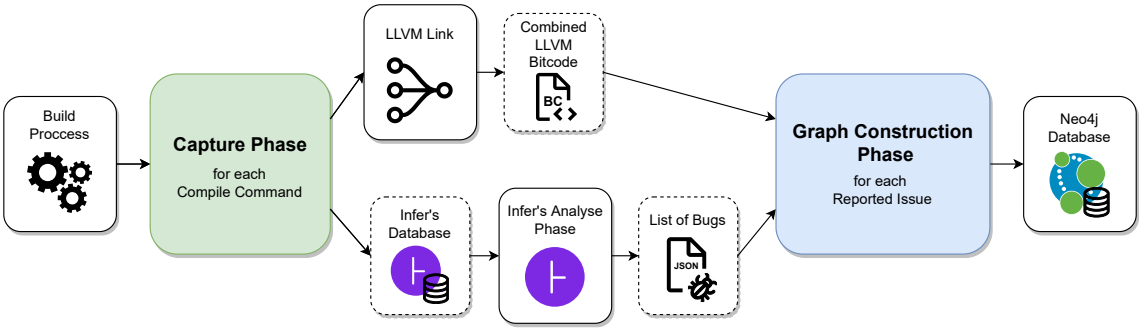

Figure 3.1: A simplified block diagram of a complete pipeline for constructing code property graphs from C source files. The dashed line indicates intermediate products. The diagram uses icon from [17].

When using a GNN to find vulnerabilities in programs, it is first necessary to convert the source code of the program into a suitable graph representation. The information in Section 2.4 shows that a suitable and frequently used representation is CPG. Each application of GNN to source code is preceded by some mechanism for constructing the type of graph used. However, the currently used graph construction approaches have a number of disadvantages, which inspired the creation of this pipeline. The three main disadvantages of existing solutions are:

1. construction of insufficient graph representation, e.g. construction of only AST [53], XFG [7] or CFG [49] (however, this paper uses a hybrid approach of GNN and RNN),

2. not considering *conditional compilation* [4, 23, 54, 60, 63],

3. inability to automatically construct graphs for arbitrary software [4, 23, 54, 60, 63].

The last two points are closely linked. The previous works, namely [4, 23, 54, 60, 63] all use the Joern tool (see Section 2.7) to construct the CPG (and its various modifications), so they are also 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 [57], 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 cannot know which part of the code to use, or what values the macros (which are defined only during compilation) have. In experiments, it was found that Joern assumes that all macros are undefined by default, and thus irretrievably loses pieces of code that did not pass `#ifdef` or `#ifndef` conditions 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.

The use case of the proposed pipeline is also subtly different from previous works. Here we need to construct graphs from the code with respect to the Infer report that has to be verified. However, this only requires the ability to slice the code according to the extracted information from the report. The program-slicing is also used in previous works. Thus, in this respect, the previous works differ only in the way the *slicing properties* are specified and extracted.

The proposed pipeline is designed to construct CPGs from software written in C and a subset of C++. The limitation for C++ is caused by the use of LLVM Slicer, the specific reasons are discussed more in Section 2.5. A simplified *block diagram* of the complete pipeline is shown in Figure 3.1. The pipeline can be divided into two main parts:

1. Capture Phase - the goal of this part is to connect to the running build process and extract the information needed to build the graphs.

2. Graph Construction Phase - from the captured information in the capture phase, a CPG graph is constructed and stored for each vulnerability found by Infer.

The output of the whole process is a set of graphs, one graph for each vulnerability found by Infer. The output graphs are sliced to contain only information useful for future classification. Graphs simplified in this way will facilitate model learning. The graphs are stored in the Neo4j graph database, which then allows the data to be further processed, displayed, edited or passed to the GNN model.

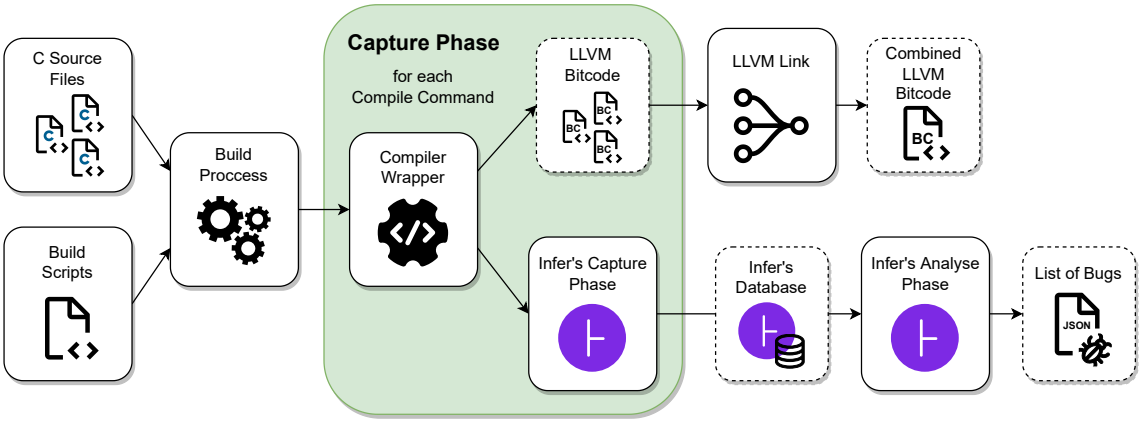

Figure 3.2: A block diagram of the capture phase of the pipeline for extracting code property graphs from C source files. The dashed line indicates intermediate products. The diagram uses icons from [17, 65].

3.1 Capture Phase

The goal of the capture phase is to connect to the running build process and capture from it the information needed for the graph construction phase (see Section 3.2). This needed information is the source files transformed into LLVM bitcode and the same source files captured by the capture phase of Infer (see Section 2.2). However, in order to obtain this information, it is necessary to obtain compilation commands from the build process. This can be done in two ways:

1. by *parsing* build scripts,
2. by capturing with the *compiler wrapper*.

Parsing build scripts is very challenging because each build system uses different *syntax* and techniques. However, there are build systems that have built-in extraction of compilation commands, such as cmake [31]. Unfortunately, this feature is far from being present in all build systems. Another problem is that there are software that do not use any of the standard build systems. Instead, they use custom scripts for compilation, *linking*, etc. These scripts can of course have any structure and *hierarchy of calls* to other scripts or tools, so it is almost impossible to statically parse compilation commands from them. The use of such scripts is quite common in SRPM packages (tool source packages for Fedora and CentOS) as seen in [1]. This pipeline is intended for use on SRPM packages and thus has to take this feature into account, see more in Section 4.5. For the reasons mentioned above, it follows that parsing build scripts, for previously unknown software, is generally not suitable for practical use.

The second and practically usable option is to create wrappers over C/C++ (or any other if needed) compilers and catch compilation commands at run-time. The design and implementation was covered in the author's previous work [1], so the principle of wrappers will only be briefly described here. Each time a compiler is called by the build system, the installed wrapper is called. On each such call, the wrapper captures its *arguments* and performs the following steps:

1. filters out options that are not compatible with Infer's internal clang compiler, which Infer uses to transform source files into SIL (see Section 2.2),
2. calls the Infer's capture phase and passes the modified compilation command to it, Infer then saves the captured source files in SIL representation to its database,
3. calls the original, unchanged, command so that the build can finish without problems.

The wrapper is designed so that even if the capture phase of the Infer fails, the original command is still executed. Failure of the Infer's capture phase will not cause the entire analysis/pipeline to crash, it may just increase the chance of generating a false positive. This error recovery is possible due to the features of Infer, which, when it does not have the necessary implementations of the functions being analyzed captured, reason that they can return any value (limited by their return type, of course). This speculation introduces a degree of *over-approximation* and thus introduces false positives. An important note is that these compiler wrappers can be (and typically are) called in *parallel*. Thus, care must be taken with possible *critical sections* such as the Infer database. Again, a description of how critical sections are handled in the wrapper can be found in [1].

For this pipeline, it is necessary to add additional wrapper functionality - generating LLVM bitcode from each captured compilation command. LLVM bitcode can be generated using the clang compiler, by inserting certain options into each captured command that does the compilation (not linking, *preprocessing* nor other things). It is needed to insert the following options in the order listed:

1. `-emit-llvm` – ensures that LLVM bitcode is generated instead of *object/binary files*,
2. `-g` – allows *backward mapping* of LLVM bitcode to the original source code,
3. `-grecord-command-line` – inserts debug information into the LLVM bitcode [55],
4. `-fno-inline-functions` – disables the use of *inline functions*,
5. `-fno-builtin` – disables inserting compiler *builtin functions*.

It is also necessary to remove the `-o` option and its value so that LLVM bitcode files are generated instead of original output. The last thing the wrapper must do is find all LLVM bitcode files on the system when it is first called after initialization. A list of these files is stored and used after the capture phase to determine which LLVM bitcode files were recently created and thus belong to the build that just finished.

After the build (capture phase of the pipeline) is complete and before the graph construction phase is started, there are three more steps to take. First, find all LLVM bitcode files that were created during the capture phase. Simply create a list of all LLVM bitcode files currently present on the system and remove any that were present before the first compilation command was executed (the previously mentioned list created at the beginning of the capture phase is useful for this). Second, merge these newly created files into a single one using the open-source[1] tool llvm-link [42]. This step is here purely for practical purposes, see Section 2.5. And the last step is to run the Infer analysis (see Section 2.2) on the

[1] **llvm-link**'s repository: `https://github.com/llvm-mirror/llvm/tree/master/tools/llvm-link`.

captured files. After the analysis is complete, Infer will generate a list of found potential vulnerabilities.

The llvm-link tool, despite its name, has nothing to do with linking as used in compilers. It is purely about linking multiple files into a single one while preserving the LLVM bitcode format. This tool was chosen because of recommendations in the DG tool documentation, see section 2.5. Also, no alternative was found and it is an official tool provided by LLVM.

3.2 Graph Construction Phase

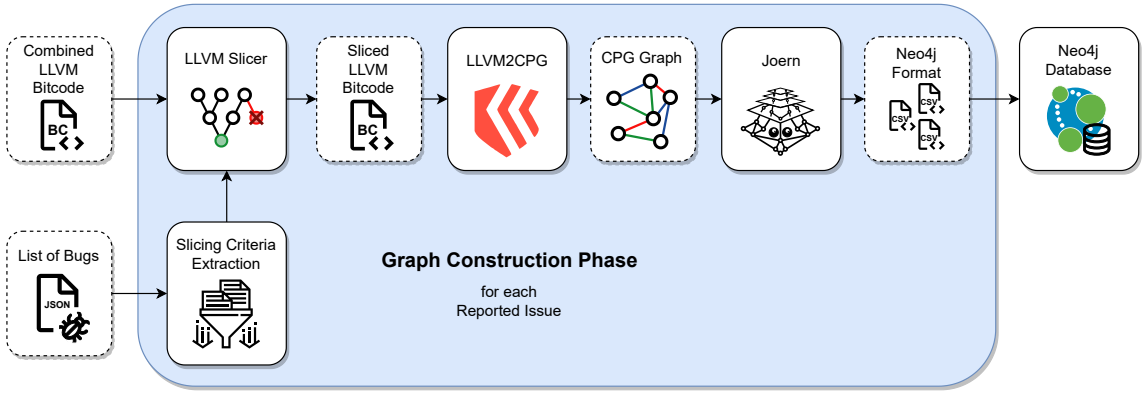

Figure 3.3: A block diagram of the graph construction phase of the pipeline for extracting code property graphs from C source files. The dashed line indicates intermediate products. The diagram uses icons from [56, 66, 67].

The input to the graph construction phase is two files (see Figure 3.3), namely a file containing all the combined LLVM bitcode and a file with a list of the vulnerabilities found by Infer. The output of this phase is a single CPG for each vulnerability. The resulting CPGs are stored in the local Neo4j graph database.

First, the information needed for the program-slicing must be extracted from each Infer report. This information includes the *entry point function* (where the program-slicing should start from), the variable/variables and their rows by which the combined LLVM bitcode will be sliced. The problem with slicing criteria extraction is that it has to be adjusted for a specific static analyzer output format (in this case for Infer). In addition, the Infer output format also varies subtly for different reported vulnerability types. Consider now two reports from Infer reported on the open-source[2] project `coreutils` – null-dereference and uninitialized value, shown in Listing 3.1 and Listing 3.2 respectively. From both reports we need to get color-coded information. The function entry point (blue) and the variable usage line (green) can be extracted from both cases (and from all other vulnerability types) in the same way. However, the variable (red) to be sliced by is only listed as part of the vulnerability-specific description. For other bug types, the way of extracting the slicing criteria can be even more different, e.g. for buffer-overrun this information has to be extracted from the `bug_trace` entry. This implies that it will be necessary to examine all Infer bug types and manually create slicing criteria extraction for each of them. An important

[2] **coreutils**'s repository: https://github.com/coreutils/coreutils.

```
{
    "bug_type": "NULL_DEREFERENCE",
    "qualifier": "pointer `new_chunk` last assigned on line 200 could be
    null and is dereferenced at line 204, column 3.",
    "severity": "ERROR",
    "line": 204,
    "column": 3,
    "procedure": "_obstack_newchunk",
    "procedure_start_line": 181,
    "file": "lib/obstack.c",
    "bug_trace": [
        ...
    ],
    ...
}
```

Listing 3.1: Null-dereference Infer report sample on the `coreutils` project.

note is that to use a different static analyzer, only the slicing criteria extraction needs to be adjusted (and of course the static analyzer needs to be run, but this is not a problem thanks to csmock, see Section 4.5).

After extracting the information for program-slicing, the LLVM slicer (described in Section 2.5) is called with the necessary parameters on the combined LLVM bitcode file (the same LLVM bitcode file is used for all vulnerabilities). The output is sliced LLVM bitcode. This is passed to the LLVM2CPG tool (described in Section 2.6), which constructs the CPG. One possible tool that can process the format from LLVM2CPG is Joern. Although Joern is a very powerful tool for various tasks in static analysis, it is only used in this pipeline as a transformer of the CPG format. In fact, Joern is able to transform CPG into `csv` format, which can be easily loaded into the Neo4j graph database. Thus, each CPG (for each vulnerability) is inserted into the Neo4j database in the last part of the pipeline. All possible node and edge properties of the resulting CPG are specified in the Joern documentation [57].

The LLVM slicer tool was selected on the recommendation of Ing. Viktor Malík. After verification, it was found that LLVM slicer meets all the requirements of the pipeline design. The tool is also under active development. An alternative to LLVM slicer is e.g. llvm-slicing[3], but this tool is no longer maintained for a long time. LLVM2CPG was chosen on the recommendation of the Joern tool documentation [57], which commonly uses LLVM2CPG and depends on it for its support of LLVM bitcode. The reason for choosing the Joern tool was that it is the main choice of most of the above mentioned articles that construct graphs for GNNs from source files. And the Neo4j database was chosen for several reasons. It is one of the most widely used graph databases and is very well *optimized* for operations on stored graphs. Another reason is the intuitive documentation[4] and also the Python driver documentation[5], which will be useful for passing data into GNN.

[3] **llvm-slicing**'s repository: https://github.com/zhangyz/llvm-slicing.
[4] **Neo4j database** documentation: https://neo4j.com/docs/.
[5] **Neo4j Python driver** documentation: https://neo4j.com/developer/python/.

```
{
    "bug_type": "UNINITIALIZED_VALUE",
    "qualifier": "The value read from n_read was never initialized.",
    "severity": "ERROR",
    "line": 412,
    "column": 15,
    "procedure": "elide_tail_bytes_pipe",
    "procedure_start_line": 247,
    "file": "src/head.c",
    "bug_trace": [
        ...
    ],
    ...
}
```

Listing 3.2: Uninitialized value Infer report sample on the `coreutils` project.

Chapter 4

Future Work

This chapter describes the future work that, together with this term project, will constitute the Master's thesis. Section 4.1 describes the automation of the proposed pipeline. Section 4.2 describes the selected dataset and its transformation to the required format. Section 4.3 describes the architecture requirements and briefly some existing architectures. Section 4.4 briefly describes the self-training technique and its use in this thesis. Section 4.5 describes the planned integration with the csmock tool. Finally, Section 4.6 describes the data that will be used to *evaluate* the accuracy of the trained model.

4.1 Automation of a Graph Construction

It is clear that the graph construction pipeline proposed in this term project must be fully automated in order to be usable. The goal of future thesis work is to detect false positives among the vulnerabilities reported by Infer on SRPM packages within the csmock tool (more in Section 4.5). Csmock is used to automatically analyze SRPM packages using various static analyzers. Infer has already been integrated into csmock, as previous work by the author [1]. Because of the integration into csmock, it is necessary that not only the pipeline, but also everything that follows it - graph preprocessing and classification using GNNs - is fully automated.

Automation is currently implemented for the capture phase, the input preprocessing construction phase and the construction phase itself, except slicing criteria extraction. This part turns out to be the most difficult part, as it requires individual approach to different types of reported vulnerabilities.

4.2 Construction of Graphs from D2A Dataset

In order to create a graph neural network that detects false positives, it must first be trained for such a task. And for that, a suitable dataset is needed. D2A [62] is a dataset created by researchers at IBM Research. The pipeline code to create it is open-source[1]. It is a dataset created from real-world open-source projects [28] namely `ffmpeg`, `httpd`, `libav`,

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

`libtiff`, `nginx` and `openssl`. D2A is created in a hybrid way combining *manual labeling* and *automatic generation*, see [62] for more details. This gives it interesting properties. Compared to manually generated datasets, it contains many more samples since it can be automatically generated. Unfortunately, however, it has lower *accuracy*. In contrast to automatically generated datasets (synthetic), it has samples from real-world software.

The reason for using D2A in future work is that it was created using Infer and thus each sample contains detailed Infer output from which slicing criteria can be extracted. Furthermore, each sample contains information about the type of vulnerability (the types are given by Infer), its location, the correctness of the report and the bug trace. However, the samples do not contain graphs but the source code of all the functions that are listed in the Infer's bug trace of a particular vulnerability. For this reason, the source code will first need to be converted to CPGs. However, this transformation will be very resource expensive, as each sample needs to be compiled and run the pipeline proposed above over it.

4.3 Graph Neural Network Architecture

The next step is to select and possibly implement a specific graph neural network architecture. An interesting note is that the eventual trained model will not depend on the programming language (due to the use of LLVM) or the static analyzer (since it is only used for extracting the slicing criteria). However, the architecture must satisfy a few requirements:

1. must respect the CPG format (see section 2.4),
2. be suitable for classification,
3. not too large to train the model using D2A and self-training (see Section 4.4).

Originally, the architecture chosen was from BGNN4VD [4], which has a CCG as input, which is very similar to a CPG, except that it uses a DFG instead of a PDG. The goal was to detect whether a function is vulnerable or not (function-level vulnerability). However, the paper does not provide, neither a link to the model, nor detailed model specifications (number of layers, parameters, used activation functions, etc.). And unfortunately, it was not even possible to contact the authors of this article.

Another interesting architecture is Devign [63]. The model input is again a combination of AST, CFG, DFG and natural code sequence (NCS). NCS is just an added set of edges that explicitly states the order of tokens in the original code. The model is again designed for function-level classification. Other interesting models can be found, e.g. in papers [7, 20].

4.4 Self-training

Self-training [10] is a relatively simple method that allows the use of *unlabeled data* in arbitrary *supervised learning* method. Thus, it is an extension of supervised learning that makes it *semi-supervised*. In principle, it is an extension of the *labeled dataset* with unlabeled samples as follows:

- Train the model on the labeled dataset and then iteratively execute:
  1. run the trained model on unlabeled data,
  2. add pseudo-labels to samples that have a high probability of correct *prediction* (the model is very confident),
  3. create a dataset by combining *labeled samples* and *pseudo-labeled samples*,
  4. train the model on the newly created dataset and jump to 1).

This technique ideally improves the *generalization* and accuracy of the model. Vulnerability detection in particular is an ideal task for self-learning, as there is a relatively small amount of good quality labeled data available and on the other hand we have a large amount of unlabeled data. The unlabeled samples can be easily and quickly created using a designed pipeline that we run (without a model) within the csmock tool on SRPM packages.

4.5 Integration with Csmock

As previously mentioned, this whole approach (graph construction pipeline and model) was designed for integration with the csmock tool [12, 13] and subsequent analysis of SRPM packages. The open-source[2] tool csmock is used to automatically run static analyses over SRPM packages (tool source packages for Fedora and CentOS operating systems). The csmock tool uses the mock tool to create an isolated and controlled environment in which a SRPM package is compiled. Static analyses are also run in this created environment. The basic principle of csmock is illustrated in Figure 4.1. The Meta Infer tool has already been integrated into csmock as part of the author's Bachelor's thesis [1].

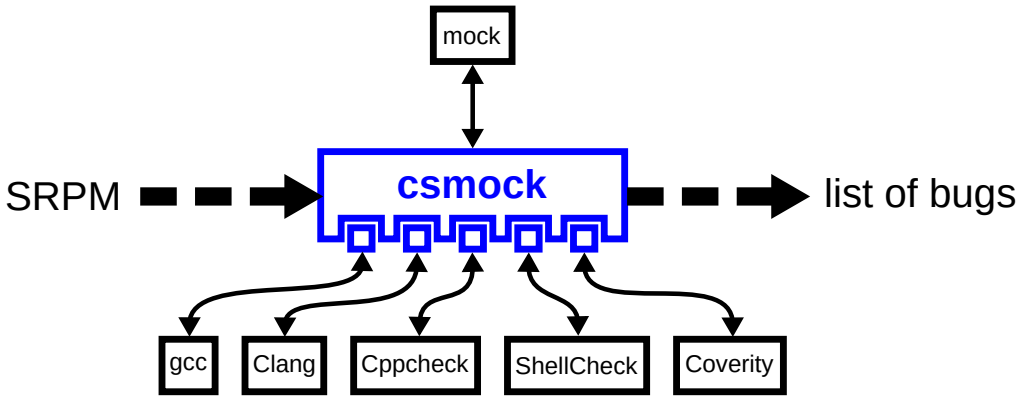

Figure 4.1: An illustration of the basic usage of the csmock tool for automated analysis of SRPM packages. The figure was taken from [13].

Integrating this work into csmock will allow automatic checking of Infer reports. Since the model will produce a soft score for each error that indicates its probability of a false positive, the check can be approached in two ways. Either a hard decision will be made using the threshold and reports that have a high probability of being false positives will be

[2] **csmock**'s repository: https://github.com/csutils/csmock.

filtered out. Or a soft score will be used to sort the reports. The idea behind this method is that no report that is potentially a true positive will be discarded, but if there is no time to check everything (which in practice there almost never is) it is possible to focus first on reports with a higher probability of being true positives according to the model. So ideally it should be possible to find most of the true positives in a first few of the reviewed reports.

4.6 Experiments

As mentioned earlier the output of this work should be applied to real-world software - SRPM packages. Ideally, experiments should be performed directly in this *target domain*. The author's previous work [1] also include the analysis of several selected SRPM packages written in C using Infer. Specifically, these packages are `hostname`, `zip`, `cswrap`, `grep`, `make`, `mlocate`, `less`, `sed`, `tree` and `psmisc`. These analyses found a number of real bugs that were reported and fixed by the developers of the respective packages. The previous work makes it possible to evaluate the success of the model directly in the target domain and on real-world data with manual labels. There are approximately 400 manually labeled Infer reports on the above-mentioned SRPM packages and approximately another 200 on other software.

Chapter 5

Conclusion

This term project first described static analysis in the context of finding vulnerabilities and the static analyzer Meta Infer. Then the basic concept of GNN was introduced, which served mainly as an introduction to the specification of GNN inputs. Then, one of the most widely used graph representations for finding vulnerabilities in source code – code property graphs – was introduced. This was followed by a description of the tools used. The core of the term project was a description of the design of a pipeline for automatic construction of code property graphs from source code. Finally, the thesis described the future steps needed to create a system for detecting false positives in Meta Infer reports.

The main contribution of this term project was the design of a pipeline for automatic extraction of code property graphs from source files written in C and a subset of C++. The goal of this design was to address the disadvantages of existing solutions, namely the disregard of conditional compilation and the inability to automatically identify source files to be analyzed. The proposed pipeline addresses both of these disadvantages thanks to its ability to connect to a running compilation process, from which all necessary information is extracted for the following construction of code property graphs. The pipeline also uses a program slicing to remove unnecessary parts of the graphs in order to simplify the resulting graph and simplify future model training. The output CPGs are also independent of the input programming language of the pipeline due to the use of LLVM IR (as a source for constructing CPGs), however pipeline is still only applicable to C/C++. And since the Meta Infer report is only used as a source of slicing criteria, the output graphs are not dependent on a particular static analyzer either.

This term project will be followed by a Master's thesis that will focus on the design and development of a system for detecting false positives from the Meta Infer static analyzer. This system will consist of two parts. The first part is the pipeline proposed in this term project, which constructs code property graphs from the source code. The second part will be a trained graph neural network model that will classify the constructed graphs into false positives and true positives. This entire detection system is being designed with respect to integration into the csmock tool, so that the model can be used to inspect reports on SRPM packages.

Bibliography

[1] 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/.

[2] Berdine, J., Calcagno, C. and O'Hearn, P. W. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: *Proceedings of the 4th International Symposium on Formal Methods for components and Objects (FMCO)*. Amsterdam, Netherlands: Springer, Berlin, Germany, November 2005, p. 115–137. DOI: 10.1007/11804192_6. ISBN 978-3-540-36750-5.

[3] 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.

[4] 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. DOI: https://doi.org/10.1016/j.infsof.2021.106576. ISSN 0950-5849. Available at: https://www.sciencedirect.com/science/article/pii/S0950584921000586.

[5] Chalupa, M. DG: A program analysis library. *Software Impacts*. Elsevier. 2020, vol. 6, p. 100038. DOI: https://doi.org/10.1016/j.simpa.2020.100038. ISSN 2665-9638. Available at: https://www.sciencedirect.com/science/article/pii/S2665963820300294.

[6] 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.

[7] 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. DOI: 10.1145/3436877. ISSN 1049-331X. Available at: https://doi.org/10.1145/3436877.

[8] Denisov, A. *LLVM meets Code Property Graphs* [online]. Low Level Bits, 23. february 2021 [cit. 2023-01-29]. Available at: https://lowlevelbits.org/llvm-meets-code-property-graphs/.

[9] 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. DOI: 10.1145/3338112. ISSN 1557-7317.

[10] Dobilas, S. *Self-Training Classifier: How to Make Any Algorithm Behave Like a Semi-Supervised One* [online]. Towards Data Science, 05. december 2021 [cit. 2023-01-29]. Available at: https://towardsdatascience.com/self-training-classifier-how-to-make-any-algorithm-behave-like-a-semi-supervised-one-2958e7b54ab7.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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. DOI: https://doi.org/10.1016/j.entcs.2008.06.039. ISSN 1571-0661. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). Available at: https://www.sciencedirect.com/science/article/pii/S1571066108003824.

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

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

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

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

[20] 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. DOI:

https://doi.org/10.1016/j.cose.2022.102715. ISSN 0167-4048. Available at: https://www.sciencedirect.com/science/article/pii/S0167404822001110.

[21] 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. DOI: 10.1145/3474369.3486866. ISBN 9781450386579. Available at: https://doi.org/10.1145/3474369.3486866.

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

[23] 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.

[24] 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. DOI: 10.1109/IJCNN55064.2022.9892280.

[25] 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. DOI: https://doi.org/10.1016/j.jnca.2021.103009. ISSN 1084-8045. Available at: https://www.sciencedirect.com/science/article/pii/S1084804521000369.

[26] 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/.

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

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

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

[30] 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. DOI: 10.1109/ICSE.2013.6606613. ISSN 1558-1225.

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

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

[33] 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. DOI: 10.3390/app10051692. ISSN 2076-3417. Available at: https://www.mdpi.com/2076-3417/10/5/1692.

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

[35] 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. DOI: 10.1109/TDSC.2021.3076142.

[36] 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. DOI: 10.1109/TDSC.2021.3051525.

[37] 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.

[38] 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. DOI: 10.1145/3133956.3138840. ISBN 9781450349468. Available at: https://doi.org/10.1145/3133956.3138840.

[39] 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. DOI: 10.48550/arXiv.2006.05405.

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

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

[42] LLVM Project. *Llvm-link - LLVM bitcode linker* [online]. LLVM Project, 2003, 2023-01-28 [cit. 2023-01-29]. Llvm-link v17.0.0. Available at: https://llvm.org/docs/LangRef.html.

[43] 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/.

[44] 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/.

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

[46] 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/.

[47] 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. DOI: 10.1109/SCAM.2013.6648191.

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

[49] Rabheru, R., Hanif, H. and Maffeis, S. A Hybrid Graph Neural Network Approach for Detecting PHP Vulnerabilities. *2022 IEEE Conference on Dependable and Secure Computing (DSC)*. 2022, p. 1–9. DOI: 10.1109/DSC54232.2022.9888816.

[50] 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. DOI: 10.1109/DSC54232.2022.9888816.

[51] 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. DOI: 10.1109/ICMLA.2018.00120.

[52] 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.

[53] 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. DOI: 10.1109/ACCESS.2022.3144598.

[54] 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.

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

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

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

[58] Tonder, R. van 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. DOI: 10.1145/3180155.3180250. ISBN 978-1-4503-5638-1.

[59] 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. DOI: 10.1109/CCWC51732.2021.9376145.

[60] 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. DOI: 10.1109/ICAIT.2018.8686548.

[61] 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. DOI: 10.1109/SP.2014.44.

[62] 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. DOI: 10.1109/ICSE-SEIP52600.2021.00020.

[63] 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.

[64] 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. DOI: 10.1109/TDSC.2019.2942930.

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

[66] Eucalyp. *Funnel*. flaticon. Available at: https://www.svgrepo.com/svg/384696/compile-compiler-script-code-config.

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